Systemarchitektur

19.04.2006

Übersicht: Prozessoren, Compiler, BS-Kern } Spezifikation, Konstruktion, Korrektheit

Benutze Mathematik

Interpretation?

X X
X X XX Ar
⋮
XXXXX XXXXX Mensch

Basis B: Anzahl der Ecken

$10 = 1 \cdot B^1 + 0 \cdot B^0$

Möglichkeit der Theorie ohne Kenntnis der Dezimalzahlen (Peano Axiome)

Unärzahlen: 11111

Addition: Strich hinter die Zahl machen

Variablen für Zeichenketten: X

Definition: (X+1) = X1

Namen:
1 Eins
(1+1) Drei
111 = (3+1) 2
1111 = (2+1) = ◎ gruft
⋮

Satz: 3+1 = 2
Satz: X+Y = Y+X

allg. Definition:

(X+0) = 0

(X+(Y+1)) = (X+Y)+1

Induktion über Y

→ Körperaxiome (Landau: Grundlagen der Arithmetik)

Setzt algebraische Notation voraus

Konstanten, Variablen

Ausdrücke (X+1)·(Y+3) −4

Identitäten $(a+b)^2 = a^2 + 2ab + b^2$

2 - - - Y
Y = 3
} 2 - - - 3 = −1

Überlagerung v. Symbolen
Prioritäten
implizite Klammerung

Abkürzungen: $e^2 = (e \cdot e)$

Identitäten: $a^2 + 2ab + b^2 = (a+b)^2$

$\Delta\ 2^{2^2} = 16$ rechtsassoziativ

gelten für allen „Einsetzungen" der Variablen

Bestimmungsgleichungen: $a + 1 = 2$ $(\Rightarrow a = 3)$ Definition von vorhin

$a = 1$ Definition aus der Schule

$\Rightarrow$ „=" ist überladen

Standartdefinition von „Einsetzungen"! Setzt alg. Notation voraus

$\Rightarrow$ für Def. der alg. Notation muß man „einsetzen" ohne algebr. Notation zu erklären

Entwicklung v. natürlicher Sprache $\rightarrow$ alg. Notation

Mengenlehre (HW-Design: Keller / Paul)

das Unerklärliche: σ b Wieso haben wir (ungefähr) die gleiche

$|\,|\,\}$ Vorstellung davon, was ein Strich ist???

Ab jetzt: Algebraische Notation und naive Mengenlehre vorausgesetzt

---

Kapitel 1: Rechnen mit boole'schen Ausdrücken

Konstanten: 0, 1

Operatoren: $\sim, \wedge, \vee, \oplus$

boole'sche Funktionen: $\sim : \{0,1\} \rightarrow \{0,1\}$

$\wedge, \vee, \oplus : \{0,1\}^2 \rightarrow \{0,1\}$

x_1	$\sim x_1$
0	1
1	0

Prioritäten: $\sim$, $\wedge$, $\vee$ (hoch $\rightarrow$ niedrig)

unär $\Leftrightarrow$ binär „Punkt vor Strich"

x_2	x_0	$x_1 \wedge x_0$	$x_1 \vee x_0$	$x_1 \oplus x_0$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Identitäten: $x_1 \wedge x_2 = x_2 \wedge x_1$ Kommutativgesetz

$(x_1 \wedge (x_2 \wedge x_3)) = ((x_1 \wedge x_2) \wedge x_3)$ Associativgesetz

$(x_1 \wedge (x_2 \vee x_3)) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ Distributivgesetz $(\cdot, +)$

$(+, \cdot)$ ✗

Beweise: Wahrheitstafeln (abhängig von Variablenanzahl)

$x_1 \vee \overline{x_1} = 1$

$x_1 \wedge \overline{x_1} = 0$

Abkürzung: $x_1 x_2 \triangleq x_1 \wedge x_2$

$\overline{x_1} \triangleq \sim x_1$

Lösen von Gleichungen: $\bar{x} = 1 \Leftrightarrow x = 0$
$\bar{x} = 0 \Leftrightarrow x = 1$

$x_1 \wedge x_2 = 1 \Leftrightarrow x_1 = 1$ und $x_2 = 1$

$x_1 \wedge \ldots \wedge x_n = 1 \Leftrightarrow \forall i \in \{1..n\} : x_i = 1$

$x_1 \vee x_2 = 1 \Leftrightarrow x_1 = 1$ oder $x_2 = 1$

$x_1 \vee \ldots \vee x_n = 1 \Leftrightarrow \exists i \in \{1..n\} : x_i = 1$

$f: \{0,1\}^n \to \{0,1\}$

gesucht: Ausdrücke mit Variablen $x_1, .. x_n$, so daß

$f(x_1, \ldots x_n) \equiv e \Leftrightarrow f(x_1 .. x_n) = 1 \Leftrightarrow e = 1$

23.04.2006

Wiederholung: Operatoren: $\neg, \wedge, \vee, \oplus$

$=$ / $\backslash$ = Bestimmungsgleichung
$\equiv$ Identität
} meistens Unterschied aus Zusammenhang klar
⇒ schreibe meistens „=" statt „≡"

Bsp: $a - 1 = 0$ gilt für ein a
$a \equiv a$ alle a

Boolesche Gleichungen lösen:

$\bar{e} = 0 \Leftrightarrow e = 1$
$\bar{e} = 1 \Leftrightarrow e = 0$

$e_1 \wedge \ldots \wedge e_n = 1 \Leftrightarrow \forall i : e_i = 1$
$e_1 \vee \ldots \vee e_n = 1 \Leftrightarrow \exists i : e_i = 1$

Definition (Schaltfunktion): $f: \{0,1\}^n \to \{0,1\}$

Einschub: M Menge

Definition: $M^n = \{(a_1 \ldots a_n) \mid a_i \in M \text{ für } i = 1..n\}$

z.B: $\{0,1\}^2 = \{(0,0), (0,1), (1,0), (1,1)\} = (00, 01, 10, 11)$

Beispiel: Addierer

a	b	c	c'	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum $s: \{0,1\}^3 \to \{0,1\}$

Carry $c': \{0,1\}^3 \to \{0,1\}$

Gegeben f: Gibt es Ausdruck e: $e \equiv f(x_1 .. x_n)$

$$e = 1 \Leftrightarrow f(x_1 .. x_n) = 1$$

Definition: e berechnet f

Satz (Darstellungssatz): $\forall f \; \exists e.\; e$ berechnet f

Beweis: Sei x Variable, $\varepsilon \in \{0,1\}$

$$x^\varepsilon = \begin{cases} x & \varepsilon = 1 \\ \bar{x} & \varepsilon = 0 \end{cases} \qquad \text{Literal}$$

Lemma: $x^\varepsilon = 1 \Leftrightarrow x = \varepsilon$

Beweis: $\varepsilon = 1$: $x^1 = 1 \Leftrightarrow x = 1$
$\varepsilon = 0$: $x^0 = \bar{x} = 1 \Leftrightarrow x = 0$ □

Sei $a = (a_1 .. a_n) \in \{0,1\}^n$.

$m(a) = x_1^{a_1} \wedge x_2^{a_2} \wedge \dots \wedge x_n^{a_n}$ Monom von a

Lemma: $m(a) = 1 \Leftrightarrow x_i = a_i$ für alle $i \in \{1..n\}$

Beweis: $m(a) = 1 \Leftrightarrow x_1^{a_1} \wedge \dots \wedge x_n^{a_n} = 1$

$\Leftrightarrow x_1 = a_1 \wedge \dots \wedge x_n = a_n$ □

Definition: $T(f) = \{a \mid f(a) = 1\}$ Träger von f

$$\boxed{d(f) = \bigvee_{a \in T(f)} m(a)}$$ vollständige disj. Normalform (DNF)

$T(f) = \emptyset \Rightarrow d(f) = 0$

Beweis (Satz): $d(f) = 1 \Leftrightarrow f(x_1 .. x_n) = 1$

$\Leftrightarrow \exists a \in T(f) : \underbrace{m(a) = 1}$

$\Leftrightarrow x = a$

$\Leftrightarrow x \in T(f)$

$\Leftrightarrow f(x) = 1$

qed

Definition: Straight line Programm

Seg.: Eingangsvariablen $\{x_1, x_2, \ldots\} \in V$

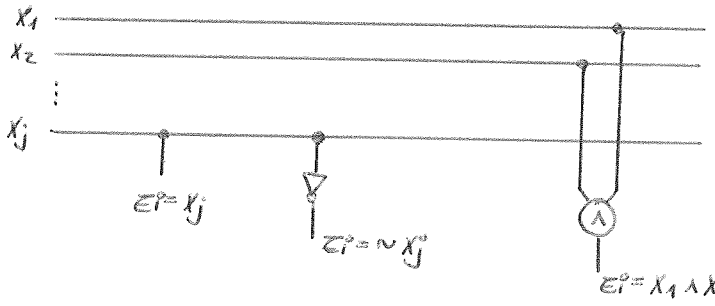
Straight Line Programm: Folge von Zuweisungen $(\varepsilon_1 .. \varepsilon_n)$ mit:

$\forall i \in \{1..n\}$ gilt:

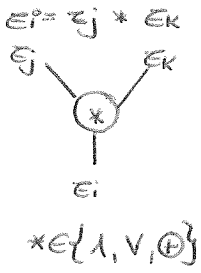
Identität:
- $\varepsilon_i \in V$ oder
- $\varepsilon_i = \sim \varepsilon_j$, mit $j < i$ oder
- $\varepsilon_i = \varepsilon_j * \varepsilon_k$, mit $j,k < i$ und $* \in \{\wedge, \vee, \oplus\}$

Verboten: $\varepsilon_i = \overline{\varepsilon_i}$

Konstruktionsvorschrift für Schaltkreise

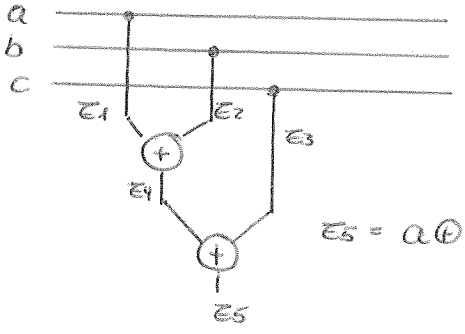


allg.:



Beispiel:

$\varepsilon_1 = a$
$\varepsilon_2 = b$
$\varepsilon_3 = c$
$\varepsilon_4 = \varepsilon_1 \oplus \varepsilon_2$
$\varepsilon_5 = \varepsilon_3 \oplus \varepsilon_4$



$\varepsilon_5 = a \oplus b \oplus c = S(a,b,c)$

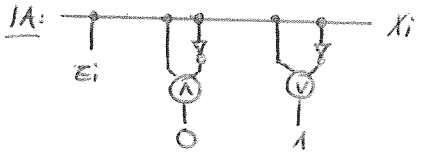
Satz: $\forall$ Booleschen Ausdrücke e $\exists$ Straight line Programm / Schaltkreis

$S(e) = (\varepsilon_1 \ldots \varepsilon_n) : \varepsilon_n = e$

Beweis: Definiere $K(e)$ = # Operatoren $(\sim, \wedge, \vee, \oplus)$: Komplexität

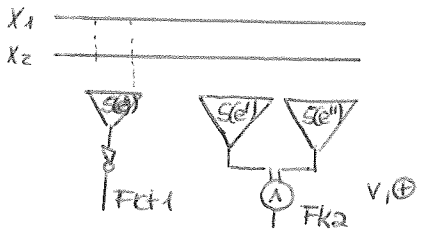
Induktion über $K(e)$

IA: $K(e) = 0$: $e = 0, 1, x_i$



IS: $K(e) = k \rightsquigarrow K(e) = k+1$: Fälle: $e = \sim e'$

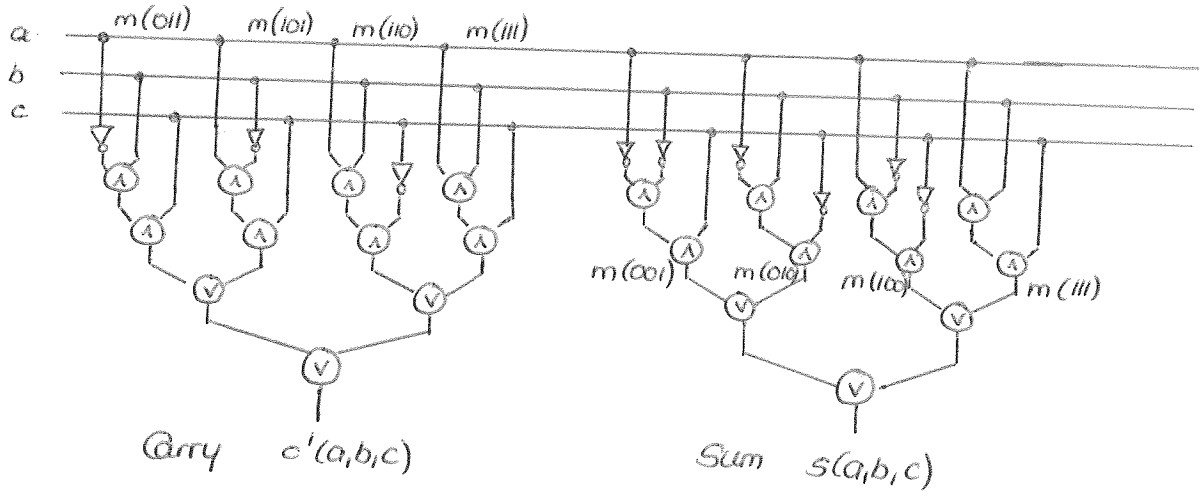
$e = e' \wedge e''$ $(\vee, \oplus)$



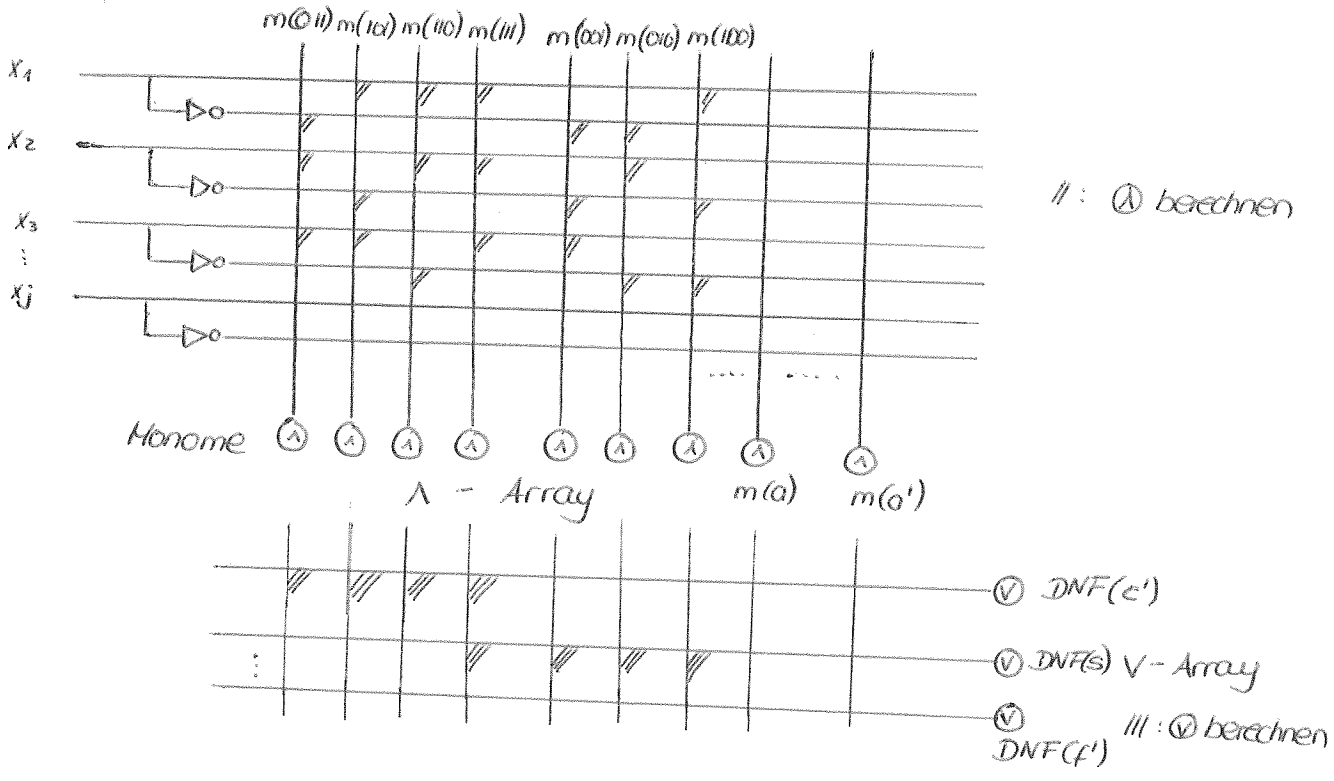
Korollar: $\forall f: \{0,1\}^n \to \{0,1\}$ $\exists$ Schaltkreis $S = (\varepsilon_1 \dots \varepsilon_m)$ (S berechnet f)

mit $\varepsilon_m \equiv f(x_n \dots x_n)$

Forme DNF in Schaltkreis um



Bei Realisierung: Verschwendung der Bauteile



geht nur für kleine n : • Einsatz in der Kontrolllogik

• PLA (programmable logic array)

$\#$ a in $T(f) \triangleq \# \{0,1\}^n = 2^n$

$f: \{0,1\}^n \to \{0,1\}$

Kapitel 2: Elementare Rechnerarithmetik

$B \in \mathbb{N}, B \neq 0, B \neq 1$ Basis der Zahlendarstellung (# der Ziffern)

$a_i \in \{0, \dots B-1\}$ Ziffer

$a = (a_{n-1} \dots a_0) = a[n-1:0]$ Ziffernfolge

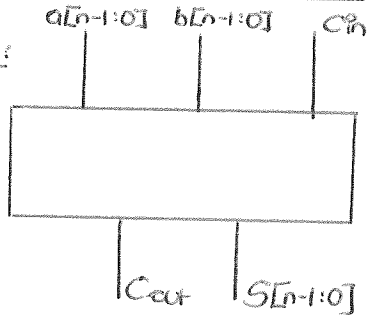
$\langle a[n-1:0] \rangle_B = \sum_{i=0}^{n-1} a_i B^i$ Zahl von a zur Basis B

Bsp: $\langle 10 \rangle_2 = 1 \cdot 2^1 + 0 \cdot 2^0 = 2$

$\langle 10 \rangle_{10} = 1 \cdot 10^1 + 0 \cdot 10^0 = 10$

Definition eines n-Bit-Addierers

Schaltkreis:



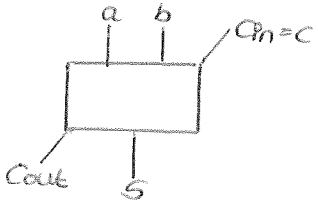
mit $\langle C_{out}, S[n-1:0] \rangle_2$

$= \langle a[n-1:0] \rangle_2 + \langle b[n-1:0] \rangle_2 + C_{in}$

$$\langle C_{out}, S \rangle = \langle a \rangle + \langle b \rangle + C_{in}$$

Volladdierer (Full-Adder) ist ein 1-Bit-Addierer

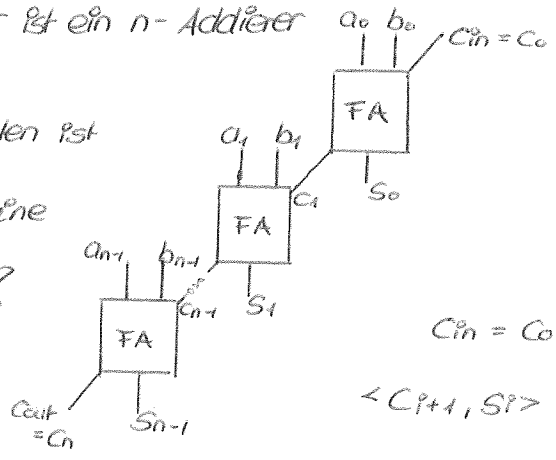
$\langle c', s \rangle = a + b + c$, $a, b, c \in \{0,1\}$ (c', s in Tabelle vorher spezifiziert)



26.04.2006

Satz: Carry-Chain-Addierer ist ein n-Addierer

Mit welchen Identitäten ist Schaltkreis (Straight Line Programm) konstruiert?



$C_{in} = C_0$

$\langle C_{i+1}, S_i \rangle = a_i + b_i + C_i$

Beispiel:

i	4	3	2	1	0	
c_i	1	1	1	0	0	$c_{in} = 0$
a_i	0	0	1	1	0	$\langle a \rangle = 6$
b_i	0	1	0	1	1	$\langle b \rangle = 11$
S	1	0	0	0	1	$\langle S \rangle = 17$ ✓

$1951 = 19 \cdot 10^2 + \underbrace{51}_{2}$

$19511 = 19 \cdot 10^3 + \underbrace{511}_{3}$

$$\langle a[n-1 \dots L, L-1 \dots 0] \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

$$= \sum_{i=L}^{n-1} a_i 2^i + \sum_{i=0}^{L-1} a_i \cdot 2^i$$

$$= \sum_{j=0}^{n-L-1} a_{L+j} \cdot 2^{L+j}$$

$$= 2^L \cdot \langle a[n-1 : L] \rangle$$

Lemma: $\langle a[n-1:0] \rangle = \langle a[n-1:L] \rangle \cdot 2^L + \langle a[L-1:0] \rangle$

Beweis (Satz): Induktion über i

Induktionsvoraussetzung: $\langle c_i, S[i-1:0] \rangle = \langle a[i-1:0] \rangle + \langle b[i-1:0] \rangle + c_{in}$

Induktionsanfang: $i = 1$: Definition von $\langle c_1, S_0 \rangle$

Induktionsschritt: $i \rightsquigarrow i+1$:

$$\langle a[i:0] \rangle + \langle b[i:0] \rangle + c_{in} =$$

$$a_i 2^i + \langle a[i-1:0] \rangle + b_i 2^i + \langle b[i-1:0] \rangle + c_{in} =$$

$$a_i 2^i + b_i 2^i + \langle a[i-1:0] \rangle + \langle b[i-1:0] \rangle + c_{in} \overset{IV}{=}$$

$$a_i 2^i + b_i 2^i + \langle c_i, S[i-1:0] \rangle \overset{\text{Lemma}}{=}$$

$$(a_i + b_i + c_i) 2^i + \langle S[i-1:0] \rangle \overset{\text{Def. Additionsschritt}}{=}$$

$$\langle c_{i+1}, S[i:0] \rangle 2^i + \langle S[i-1:0] \rangle =$$

$$\langle c_{i+1}, S[i:0] \rangle$$ qed

Kosten: $K(S)$ = # Gatter in S

Verzögerungszeit

Länge m

Pfad in S: Folge von Gattern $e_1 \to e_2 \dots \to e_m$

$\forall i$: $Out(e_i) = In(e_{i+1})$

Tiefe $T(S)$: max. Länge der Pfade in S

Sei S_n eine Länge Carry-Chain Addierer (n Bits)

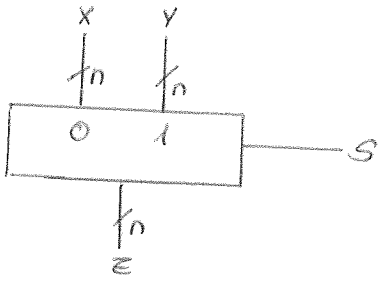
$K(S_n) = n \cdot K(FA)$ billig

$T(S_n) = n \cdot T(FA) \leq 3n$ langsam

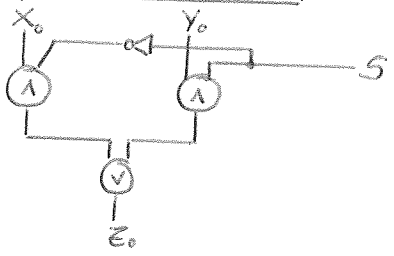
Schnelle Addierer

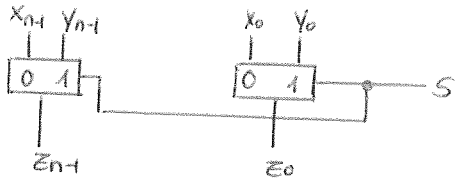
Definition: n-Bit Multiplexer (MUX)

$$z = \begin{cases} x: & s = 0 \\ y: & s = 1 \end{cases}$$



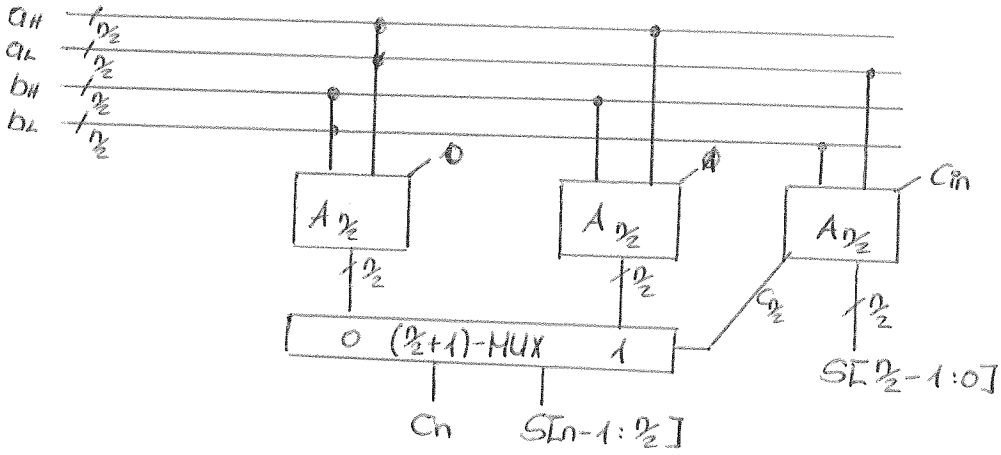
Kostenfunktion: 1-MUX





$T(n\text{-MUX}) = 3$

$A_1 = FA$ $\qquad A_{\frac{n}{2}} \to A: \quad a_H = a[n-1 : \frac{n}{2}]$ $\quad a_L = a[\frac{n}{2}-1 : 0]$



Def. $T_n = T(A_n)$
$T_1 = T(FA) = 3$

$T_n \leq T_{\frac{n}{2}} + 3$

Differenzengleichung:

Annahme: $T_1 = 3$ $\qquad n = 2^k, k \in \mathbb{N}$
$T_n = T_{\frac{n}{2}} + 3$ $\qquad k = \log_2 n$

$$T_n = T_{\frac{n}{2}} + 3 = T_{\frac{n}{4}} + 3 + 3 = T_{\frac{n}{8}} + 3 + 3 + 3 \overset{\text{Vermutung}}{=} T_{\frac{n}{2^j}} + 3 \cdot j$$

Ende $j = k$: $T_{\frac{n}{2^k}} + k \cdot 3 = T_1 + k \cdot 3 = T(FA) + 3 \log_2 n$

E.E.: $T_n = T(FA) + 3 \log_2 n$

$n = 1$: $T_1 = \underbrace{T(FA)}_{3} + \underbrace{3 \log_2 1}_{0} = 3$

$\frac{n}{2} \rightsquigarrow n$: $T_n = T_{\frac{n}{2}} + 3$ Diffgleichung

$(k \rightsquigarrow k+1)$ $\qquad = T(FA) + 3 \log_2 (\frac{n}{2}) + 3$ (IV)

$= T(FA) + 3 \log_2 n - 3 + 3 = T(FA) + 3 \log_2 n$

qed

$\bar{x}[n-1:0] = (\bar{x}_{n-1}, \dots, \bar{x}_0)$

Subtraktionsalgorithmus

Sei $x, y \in \{0,1\}^n$ und $\langle x \rangle \geq \langle y \rangle$

$$\langle x \rangle - \langle y \rangle = \langle x \rangle + \langle \bar{y} \rangle + 1 \mod 2^n$$ (letzten Übertrag wegwerfen)

03.05.2006

Exkurs: Modulo-Rechnen

$\equiv (\mod k)$ heißt Äquivalenzrelation

$\{0..k-1\}$ Repräsentantensystem

Definition: $x \mod k \in \{0..k-1\}$ heißt Repräsentant der Äquivalenzklasse von x bzgl. $\equiv \mod k$ in $\{0..k-1\}$

Satz: $x \mod k$: Rest der ganzzahligen Division von x durch k $(\text{Rest}(x,k) = \beta \in \{0..k-1\})$

$$x = k\alpha + \beta \quad , \quad \alpha \in \mathbb{Z}$$

Lemma 1: $x \equiv y \mod k$, $x \in \{0..k-1\}$ $\Rightarrow x = y \mod k$ Beweis: Übung

Lemma 2: $k = 2^n$, $m > n$

$$\langle a[m-1:0] \rangle \mod 2^n = \langle a[n-1:0] \rangle$$

Beweis: $\langle a[m-1:0] \rangle = \langle \underbrace{a[m:n]}_{\in \mathbb{N}} \rangle \cdot 2^n + \langle a[n-1:0] \rangle$

$$\Rightarrow \langle a[m-1:0] \rangle \equiv \langle a[n-1:0] \rangle \mod 2^n$$

$$\langle a[n-1:0] \rangle \in \{0..2^n-1\}$$

$\overset{\text{Lemma 1}}{\Rightarrow}$ Behauptung

Subtraktionsalgorithmus: $x, y \in [0,1]^n$, $\langle x \rangle \geq \langle y \rangle$

$$\Rightarrow \langle x \rangle - \langle y \rangle \in \{0..2^n-1\}$$

Zeige: $\langle x \rangle - \langle y \rangle \equiv \langle x \rangle + \langle \bar{y} \rangle + 1 \mod 2^n$

$\overset{\text{Lemma 1}}{\Rightarrow}$ $\langle x \rangle \geq \langle y \rangle \Rightarrow \langle x \rangle - \langle y \rangle = \underbrace{\langle x \rangle + \langle \bar{y} \rangle + 1 \mod 2^n}_{= z[n:0]}$

$\overset{\text{Lemma 2}}{\Rightarrow}$ $\langle x \rangle - \langle y \rangle = \langle z[n-1:0] \rangle$

Definition: $x \in [0,1]^n$

$$[x] = -x_{n-1} 2^{n-1} + \langle x[n-2:0] \rangle \quad \in \bar{T}_n = \{-2^{n-1}, \dots 2^{n-1}-1\}$$

heißt two's complement Darstellung (der Längen) von $[x]$

Lemma 3: $[x] < 0 \iff x_{n-1} = 1$

Beweis: $x_{n-1} = 0 \qquad \langle x[n-1:0] \rangle \geq 0$ $\qquad x_{n-1}$: Sign-bit

$$[x] \leq 2^{n-1} + (2^{n-1}-1)$$

Lemma 4: $\langle a \rangle = [0a]$

Beweis: $a \in [0,1]^n$

$$[0a] = -0 \cdot 2^n + \langle 0[n-1:0] \rangle = \langle a \rangle$$

Lemma 5: $[x] = [x_{n-1}\, x]$, $x \in \{0,1\}^n$ (Sign Extension Lemma)

Beweis: $[x_{n-1}\, x[n-1:0]] = -x_{n-1} \cdot 2^n + \langle x[n-1:0] \rangle$

$$= -x_{n-1} \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \langle x[n-2:0] \rangle$$

$$= -2^{n-1} x_{n-1} + \langle x[n-2:0] \rangle = [x]$$

Lemma 6: $\langle x \rangle \equiv [x] \bmod 2^n$

Beweis: $[x] - \langle x \rangle = -2^{n-1} x_{n-1} + \langle x[n-2:0] \rangle$

$$-2^{n-1} x_{n-1} + \langle x[n-2:0] \rangle$$

$$= -2^n x_{n-1} \equiv 0 \bmod 2^n$$

Lemma 7: $-[x] = \bar{x} + 1$

Beweis: $[\bar{x}] = -\overline{x_{n-1}} \cdot 2^{n-1} + \langle \bar{x}[n-2:0] \rangle$ $\qquad y \in \{0,1\}$ $\bar{y} = 1-y$

$$= -(1-x_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1-x_i) \cdot 2^i$$

$$= x_{n-1} \cdot 2^{n-1} - \langle x[n-2:0] \rangle - 2^{n-1} + \sum_{i=0}^{n-2} 2^i$$

$$= -[x] - 1$$

Beweis (Subtraktion): $\langle x \rangle - \langle y \rangle = \langle x \rangle - [0y]$ Lemma 4

$$= \langle x \rangle + [1\bar{y}] + 1 \quad \text{Lemma 7}$$

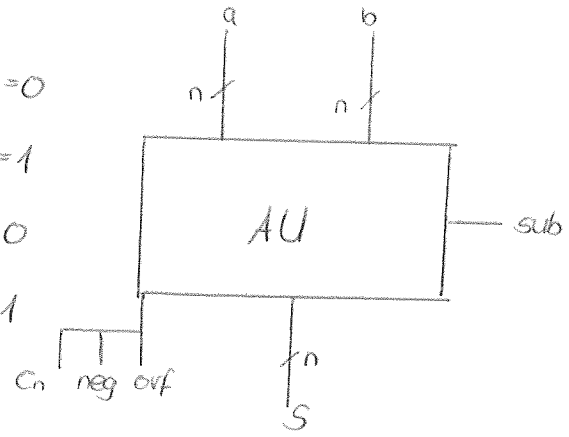
$$= \langle x \rangle - 2^n + \langle \bar{y} \rangle + 1$$

$$\equiv \langle x \rangle + \langle \bar{y} \rangle + 1 \bmod 2^n$$

qed

Definition: AU (arithmetic unit)

$$\langle S \rangle = \begin{cases} \langle a \rangle + \langle b \rangle \bmod 2^n & sub = 0 \\ \langle a \rangle - \langle b \rangle \bmod 2^n & sub = 1 \end{cases}$$



Exaktes Ergebnis: $res = \begin{cases} [a]+[b] & sub=0 \\ [a]-[b] & sub=1 \end{cases}$

$neg = 1 \Leftrightarrow res < 0$

$ovf = 1 \Leftrightarrow res \notin T_n$

Bemerkung: $res \in T_{n+1}$

$res \in [-2^n, \dots 2^n - 2\,] \subset T_{n+1}$

$[a], [b] \in T_n = \{-2^{n-1}, \dots 2^{n-1}-1\}$

08.05.2006

Satz: 1) $ovf = 1 \Leftrightarrow c_n \neq c_{n+1}$

2) $ovf = 0 \Leftrightarrow [S] = res$

Beweis: zu 1) $a + d + c_n =$

$-a_{n-1}2^{n-1} - d_{n-1}2^{n-1} + \langle a[n-2:0] \rangle + \langle d[n-2:0] \rangle + c_n =$

$-a_{n-1}2^{n-1} - d_{n-1}2^{n-1} + \langle c_{n-1}, S[n-2:0] \rangle =$

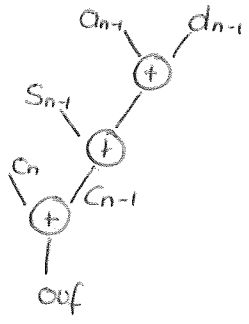
$(-c_{n-1}2^{n-1} + c_{n-1}2^{n-1})$

$-\langle c_{n-1}, s_{n-1} \rangle 2^{n-1} + 2c_{n-1}2^{n-1} + \langle S[n-2:0] \rangle =$

$-2^n c_n + 2^n c_{n-1} + [S[n-1:0]]$

Fall 1: $c_n = c_{n-1}$ ✓

Fall 2: $c_n \neq c_{n-1}$



a) $c_n = 1, c_{n-1} = 0$: $-2^n + \underbrace{[S[n-1:0]]}_{\leq 2^{n-1}} = [a]+[d]+T_{in} \leq 2^{n-1}-1 \notin T_n$

b) $c_n = 0, c_{n-1} = 1$: $[a]+[d]+c_n = 2^n + \underbrace{[S[n-1:0]]}_{\geq -2^{n-1}} \geq 2^{n-1} \notin T_n$

$[d] + c_i = \begin{cases} [d] & sub = 0 \\ -[d] & sub = 1 \end{cases}$

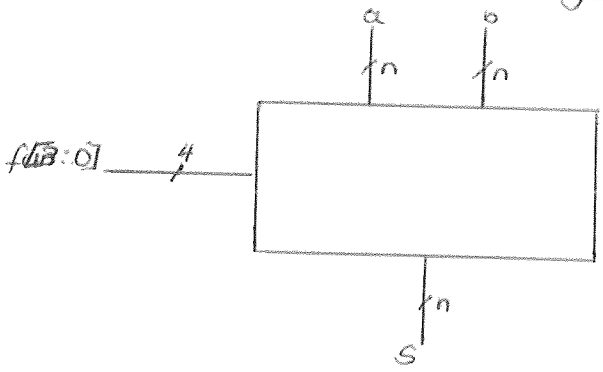
c_{n-1}: nicht in jedem Addierer berechnet

$s_{n-1} = a_{n-1} \oplus d_{n-1} \oplus c_{n-1}$

$c_n \neq c_{n-1} \Leftrightarrow c_n \oplus c_{n-1} = 1$

od 2) $[a] + [d] + c_{in} = [a_{n-1}\, a] + [d_{n-1}\, d] + c_{in} \in T_{n-1}$

Definition: n-Arithmetic Logic Unit (n-ALU)



Notation:

$a \circ b \triangleq (a_{n-1} \circ b_{n-1}, \ldots a_0 \circ b_0)$

$0^n \triangleq \underbrace{0, \ldots 0}_{n \text{ Nullen}}$ analog 1^n

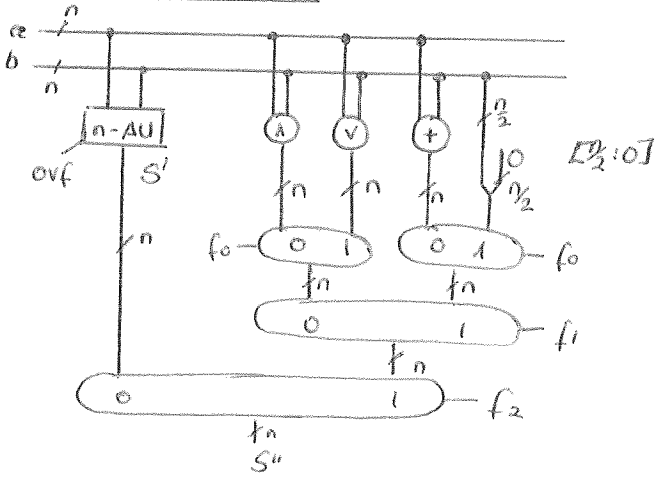
Funktion

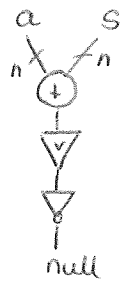
f[3:0]	s	ovf
0 0 0 0	a +n b	ovf
0 0 0 1	a +n b	0
0 0 1 0	a -n b	ovf
0 0 1 1	a -n b	0
0 1 0 0	∧	
0 1 0 1	∨	
0 1 1 0	⊕	
0 1 1 1	$b[n/2:0]0^{\frac{n}{2}}$	

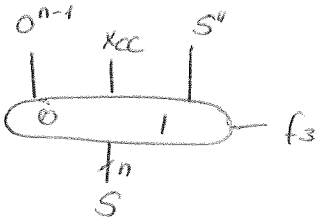
f[3:0]	xcc (a, b, f)	
	< = >	
1 000	0	
1 001	[a] > [b]	pos
1 010	[a] = [b]	null
1 011	[a] ≥ [b]	
1 100	[a] < [b]	neg
1 101	[a] ≠ [b]	
1 110	[a] ≤ [b]	
1 111	1	

xcc ≙ fixed point condition code

Konstruktion







→ neg aus AU

→ null $\left(\overline{\bigvee_i (a_i \oplus b_i)}\right)$

→ pos = $\overline{\text{null}} \wedge \overline{\text{neg}}$

→ xcc = (f2 ∧ neg) ∨ (f1 ∧ null) ∨ (f0 ∧ pos)

Kapitel 3 : Prozessor

Definition: Mathematische Maschine (Modell für schriftweises Rechnen)

$M = (K, K_0, \delta)$ mit K: Menge der Konfigurationen
$K_0 \in K$: Startkonfiguration
δ : Übergangsfunktion

Definition: Rechnung

$\forall i : k_{i+1} = \delta(k_i)$

Iteration: $\delta^n(k) = \delta(\delta^{n-1}(k))$ n-Schritte

$\delta^0(k) = k$

Beispiel: 1) **Hardware : Konf h**

h^0 nach reset
h^t während Takt t

2) **DLX : Konf cl**

cl^i vor Ausführung der i. Instruktion

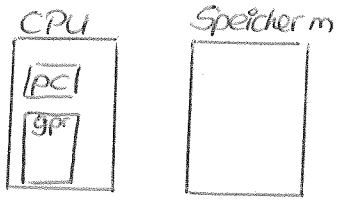
Spezifikation DLX Instruktionssatz

11.05.2006

durch mathematische Maschine $M = (K, K_0, \delta)$

$c \in K : \delta(c) = c'$ Ausführung einer Instruktion

c: „benutzersichtbare" Daten im Prozessor



$c = (c.pc, c.gpr, c.m)$

$c.m : \{0,1\}^{32} \to \{0,1\}^8$

$c.m(a) = x$ a = Adresse, x = Datum

general purpose register : $c.gpr : \{0,1\}^5 \to \{0,1\}^{32}$

$c.gpr(0^5) = 0^{32}$ Konstante 0^{32} immer verfügbar

$c.pc \in \{0,1\}^{32}$ **program counter**, zeigt im Speicher auf nächste auszuführende Instruktion $\in \{0,1\}^{32}$

$m : \{0,1\}^{32} \to \{0,1\}^8$ $d = \mathbb{N}$, $a \in \{0,1\}^{32}$

$m_d(a) = m(\underbrace{a +_{32} d_{32} -_{32} 1_{32}}_{\text{dim } (<a>-d-1)}) \circ \ldots \circ m(a)$ **Little endian**

Konvention zur Notation

Komponenten x in Konfiguration c: $c.x$ record notation

alle anderen aus c definierbaren Größen: $f(c)$ Funktionsnotation

$I(c) = c.m_4(c.pc) \in \{0,1\}^{32}$ in Konf. c auszuführende Instruktion

Instruktionsformate (siehe alle Skripte)

$Htype(c) = (c.pc(c) = 0^6) = \overline{I(c)}_{31} \wedge \overline{I(c)}_{30} \wedge \ldots \wedge \overline{I(c)}_{26}$

$jtype(c) = (c.pc(c)) \in \{000010, 000011, 111110, 111111\}$

$itype(c) = \overline{Htype(c)} \vee jtype(c)$

$RS1(c) = I(c)[25:21]$

$RD(c) = \begin{cases} I(c)[15:11] & : Htype(c) \\ I(c)[20:16] & : sonst \end{cases}$

$RS2(c) = I(c)[20:15]$

~~fct~~ $fu(c) = I(c)[5:0]$

$imm(c) = \begin{cases} I(c)[25:0] & : jtype(c) \\ I(c)[15:0] & : sonst \end{cases}$

$sxtimm(c) = \begin{cases} (I(c)_{25})^6 \; I(c)[25:0] & : jtype(c) \\ (I(c)_{15})^6 \; I(c)[15:0] & : sonst \end{cases}$ $\in \{0,1\}^{32}$

Lemma: $[sxtimm(c)] = [imm(c)]$

Instruction decoding

$lw(c) = (opc(c) = 100011)$

Semantik: $Lw(c) \rightarrow d^2(c) = c'c \ldots$ auf Speicher zugreifen an Adresse

$ea(c) = c.gpr(RS1(c)) +_{32} sxtimm(c))$ Adressierungsmode "register imm."

$RS1(c) = 0^5$

$c.gpr(RS1(c)) = 0^{32}$ immediate

$c'.gpr = \begin{cases} c.m_4(ea(c)) & x = RD(c) \\ c.gpr(x) & sonst \end{cases}$ Speicherinhalt in gpr speichern

$c'.m = c.m$

$c'.pc = c.pc +_{32} 4_{32}$

$sw(c) = (opcode(c) = 101011)$

$c'.m_4(ea(c)) = c.gpr(RD(c))$ ⚠ D NICHT Destination (sondern Quelle)

$c'.m(x) = c.m(x) \quad x \notin \{ea(c) \ldots ea(c +_{32} 3_{32})\}$

$c'.gpr = c.gpr \qquad c'.pc = c.pc +_{32} 4_{32}$

$compi(c) = (I(c)[31:30] = 01)$ compute immediately

$compr(c) = rtype \wedge (I(c)[5:4] = 00)$ compute register

$$aluf(c) = \begin{cases} I(c)[3:0] & : rtype(c) \\ I(c)[29:26] & : sonst \end{cases}$$ alu function

$lop(c) = c.gpr(RS1(c))$

$$rop(c) = \begin{cases} c.gpr(RS2(c)) & : rtype \\ sxtimm(c) & : sonst \end{cases}$$ left/right operand

$comp(c) = compr(c) \vee compi(c)$ compute

$$c'.gpr(x) = \begin{cases} aluop(lop(c), rop(c), aluf(c)) & : x = RD(c) \wedge x \neq 0^5 \\ c.gpr & : sonst \end{cases}$$

$c'.m = c.m$

$c'.pc = c.pc +_{32} 4_{32}$

jr(c) → Jump register

$c'.pc = c.gpr(RS1(c))$ $c'.m = c.m$ $c'.gpr = c.gpr$

jal(c) → Jump and link (procedure call)

$c'.pc = c.gpr(RS1(c))$

$$c'.gpr(x) = \begin{cases} c.pc +_{32} 4_{32} & : x = 1^5 \\ c.gpr(x) & : sonst \end{cases}$$

15.05.06

Kontrolloperationen:

$I[31:26]$

1101	00	beqz	$PC = PC + (RS1 = 0 ? \langle sxtimm : 4 \rangle)$	branch equal zero
	01	bnez	$PC = PC + (RS1 \neq 0 ? \langle sxtimm : 4 \rangle)$	branch not equal zero
	10	jr	$PC = RS1$	jump register
	11	jal	$PC = RS1, R31 = PC + 4$	

$A(c) = c.gpr(RS1(c))$

⇒ $taken(c) = (beqz(c) \wedge A(c) = 0^{32} \vee bnez(c) \wedge \overline{A(c) = 0^{32}})$

Prädikat: $Aeqz(c) = (A(c) = 0^{32})$

$branch(c) = beqz(c) \vee bnez(c) = (I(c)[31:27] = 11010)$

$taken(c) = branch(c) \wedge (\overline{I(c)}_{26} \wedge Aeqz(c) \vee I(c)_{26} \wedge \overline{Aeqz(c)})$

$= branch(c) \wedge (I(c)_{26} \oplus Aeqz(c))$

$jump(c) = j(c) \vee jal(c) \vee jr(c) \vee jalr(c)$

$control(c) = branch(c) \vee jump(c)$

$jbtaken(c) = btaken(c) \vee jump(c)$

$$btarget(c) = \begin{cases} c.pc +_{32} \langle sbimm(c) \rangle & : branch(c) \vee j(c) \vee jal(c) \\ A(c) & : jr(c) \vee jalr(c) \end{cases}$$

$$c'.pc = \begin{cases} c.pc +_{32} 4_{32} & : jbtaken(c) \\ btarget(c) & : jbtaken(c) \end{cases}$$

Spezifikation

$$c'.m_4(ea(c)) = c.gpr(RD(c)) : sw(c)$$

$$c'.m(x) = c.m(x) : sw(c) \vee sw(c) \wedge x \notin \{ea(c), \ldots ea(c) +_{32} 3_{32}\}$$

$$c'.gpr(x) = \begin{cases} 0^{32} & : x = 0^5 \\ c.m_4(ea(c)) & : lw(c) \wedge x = RD(c) \wedge x \neq 0^5 \\ aluop(lop(c), rop(c), aluf(c)) & : (comp(c) \vee compi(c)) \wedge x = RD(c) \wedge x \neq 0^5 \\ c.pc +_{32} 4_{32} & : (jal(c) \vee jalr(c)) \wedge x = 1^5 \\ c.gpr(x) & : sonst \end{cases}$$

Exkurs: Assemblerprogrammierung

17.05.2006

Arithmetik: 32-bit binär $(\mathbb{Z}^{32}, +_{32}, n_{32})$

$c^0.pc = 0$

$b < 2^{15} - 4$, $b + 4 \in T_{32}$

$m_4(b) = n$ anfangs: c^0

$c^\tau m_4(b+4) = \sum_{i=1}^{n} i$ schließlich: c^τ

Schleife: vom $(j+1)$ten Durchlauf der Schleife $(c^{s(j)})$

$$S(j) = n + (n-1) + (n-2) \ldots + (n-j) = c^{(s(j))}.gpr(2)$$

$$n - j = c^{s(j)}.gpr(1)$$

I(j) Induktionsbehauptung (Schleifeninvariante)

Programm:

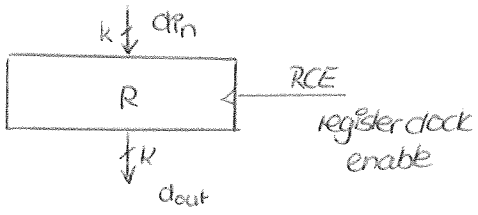
$S(j+1) = S(j) + 4 = 4j + 2$

```
0:  gpr(1) = n                 gpr(1) = m4(gpr(0)+b)        lw   0 1 b
4:  gpr(2) = gpr(4)            gpr(2) = gpr(1) + gpr(0)     add  1 0 2
                               (Pn c^z gilt I(c)^(z) s(z) = 2

8:  gpr(1) = gpr(1) - 1                                     sub  1 1 1
12: if (gpr(1) = 0) goto 4     PC(gpr(1)=0 ? PC+x = PC+4)   beqz   1 x
16: gpr(2) = gpr(2) + gpr(1)                                add  1 2 2
20: goto 8                     PC = PC - 12                 j    -12
24: m4(b+4) = gpr(2)           m4(gpr(0)+b+4) = gpr(2)      sw   0, 2, b+4
```

Prozessorbau: Erweiterung des HW-Modells

n-Bit-Register:



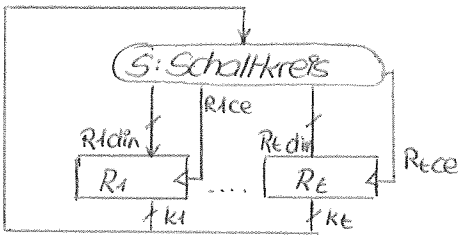
Hardwarekonfiguration h:

$h = (\ldots h.R \ldots) \quad h.R \in \{0,1\}^k$

$h = (h.R_1, \ldots h.R_t)$

Registeroutput in Konfiguration h:

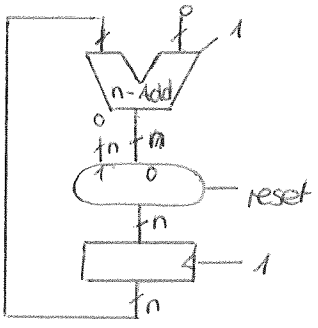
$R_{dout}(h) = h.R$



h legt alle Inputs von S: $R_i dout(h)$, $i = 1..t$ fest

⇒ Registerinputs $R_i din(h)$, $R_i ce(h)$ sind definiert durch S berechnete Funktion mit Input $R_1 dout(h), \ldots R_t dout(h)$

Beispiel: Zähler



$h = (h.R)$

reset in Takt t: (h^t) $\quad reset^{-1} = 1$, $reset^t = 0$, $t \geq 0$

Ziel: $h^0.R \in 0^n$

$h^t.R = bin_n(t) \bmod 2^n = t_n$

$$h^{t+1}.R = \begin{cases} R_{din}(h^t) & : ce(h^t) = 1 \\ h^t.R & : ce(h^t) = 0 \end{cases}$$

Im Bsp: $h^t.R = t_n$

$\Rightarrow S(h^t) = (t+1)_n$ Definition Addierer

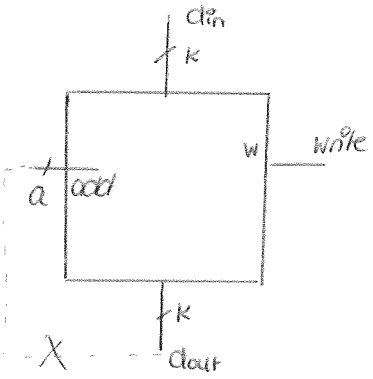
$R_{din}(t) = S(h^t)$ Definition Mux, $reset^t = 0, t \geq 0$

$R.ce^t = 1$ für alle t

$h^{t+1}.R = (t+1)_n$

□ Korrektheitsbeweis

Speicher, $2^a \times k$ RAM (random access memory)



verbieten, da Adresse nicht mehr definiert wär

$h.S: \{0,1\}^a \to \{0,1\}^k$ (Komponente der Konfiguration)

$S_{dout}(h) = h.S(S_{adr}(h))$

$$h^{t+1}.S(x) = \begin{cases} S_{din}(h^t) : x = S_{adr}(h^t) \wedge SW(h^t \\ h^t.S(x) : sonst \end{cases}$$

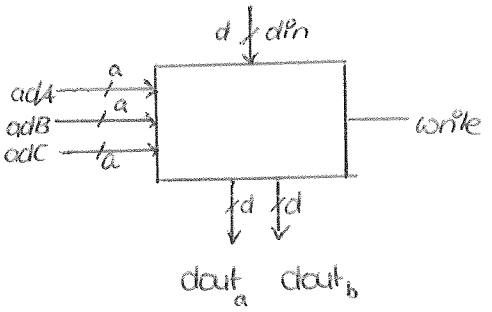
$x \in \{0,1\}^a$ Adresse

22.05.2006

DLX Konfiguration $c^0, c^1, c^2 \ldots$ $c^{i+1} = \delta_D(c^i)$ δ_D: Instruktionssatz

HW Konfiguration $h^0, h^1, h^2 \ldots$ $h^{i+1} = \delta_H(h^i)$ δ_H: HW-Übergangsfunktion

Definition: 3-Port-RAM



$d: \{0,1\}^a \to \{0,1\}^d$

$dout\text{-}A(h) = h.S(adrA(h))$
$dout\text{-}B(h) = h.S(adrB(h))$

$$h'.S(x) = \begin{cases} din(h) : w(h) = 1 \wedge x = adrC(h) \wedge x \neq 0^a \\ h.S(x) : sonst \end{cases}$$

$h.S(0^a) = 0^d$

Wann ist die Hardware korrekter DLX Processor?

=> Hardware simuliert Processor.

Hardware Konfiguration

$h = (h.IR, h.gpr, h.pc, h.m, h.ex)$

$h.IR \in \{0,1\}^{32}$ Instruktionsregister

$h.gpr$: 3-port RAM $2^5 \times 32$

$h.m$: 3-port RAM $2^{30} \times 32$

$h.ex \in \{0,1\}$ execute bit

$h.pc : \{0,1\}^{32} \to \{0,1\}^{32}$ program counter

Plan: Eine DLX Instruktion wird durch 2 Takte simuliert:

fetch ($h^{2i}.ex = 0$) : Laden von $I(c^i)$ in $h^{2i+1}.IR$
execute ($h^{2i}.ex = 1$) : Ausführung

Satz: Processor - Korrektheit

Behauptung: $\forall i: h^{2i}.pc = c^i.pc \qquad h.pc \in \{0,1\}^{32}$

$h^{2i}.gpr = c^i.gpr$

$h^{2i}.m = c^i.m$

Voraussetzung: $reset^{-1} = 1$
$reset^t = 0 \quad : t \geq 0$

Va) $h^{2i}.m(a) = c^i.m_4(a00)$ // unrealistisch stark ⇒ Real.: Va) gilt nicht

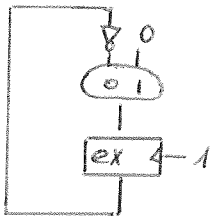
Vb) $h^0.m\,\xi = c^0.m$
gilt für Teil von RAM:
ROM, read only memory

Software Bedingung:
Register erst lesen, nachdem sie geschrieben wurden

Software Bedingung: Alignment

alle $<c^i.pc>, <eq(c^i)>$ müssen durch 4 teilbar sein.

Execute Schaltkreis:



Lemma: $h^t.ex = 1 \Leftrightarrow t$ ungerade

Lemma: $h^{2i+1}.IR = I(c^i)$

Beweis: Teil des Induktionsschritts des Satzes

$h^{2i}.pc = c^i.pc$ IV

$h^{2i}.ex = 0$ Lemma

$ad(h^{2i}) = h^{2i}.pc\,[31:2]$

$= c^i.pc\,[31:2]$

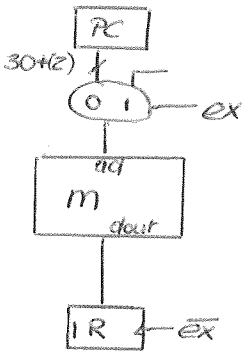
$Dout(h^{2i}) = h^{2i}.m(c^i.pc[31:2])$

$= c^i.m_4(c^i.pc[31:2]00)$ IV

$= c^i.m_4(c^i.pc) = I(c)$

$h^{2i+1}.IR = Dout(h^{2i}) = I(c^i)$

IR:



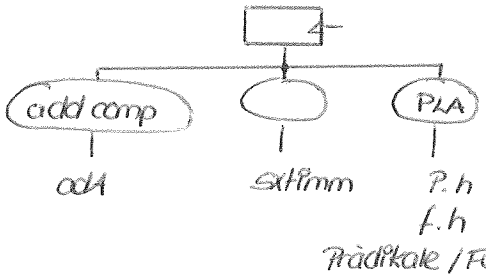
HW : $gpr(w) = ex\ 1$
$m.w = ex\ 1$

$gprw(h^{2i}) = m.w(h^{2i}) = 0$

$\Rightarrow h^{2i+1}.gpr = h^{2i}.gpr \overset{IV}{=} c^i.gpr$

$h^{2i+1}.m(x) = h^{2i}.m(x) \overset{IV}{=} c^i.m_4(x00)$

Instruktionsdecoder



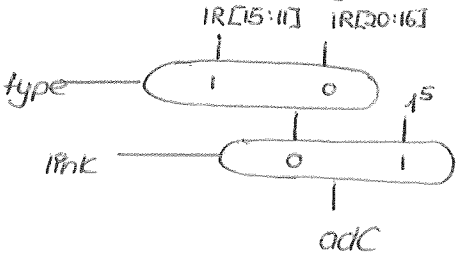
$p.h$ und $p(c)$:

$p(c) = p'(I(c))$ z.B. $lw(c), addi(c)$

$p.h = p'(IR)$

Lemma: $p_h(h^{2i+1}) = p(c^i)$
$f_h(h^{2i+1}) = f(c^i)$

Adreßberechnung



$lImk(c) \rightarrow adC = 1^5$

$\overline{lImk(c)} \rightarrow adC(h^{2i+1}) = rd(c^i)$

$adA = IR[25:21]$ $\quad adA(h^{2i+1}) = RS1(c^i)$
$adB = IR[20:16]$ $\quad adB(h^{2i+1}) = RS2(c^i)$

$A(h^{2i+1}) = \underbrace{h^{2i+1}.gpr}_{c^i.gpr}(\underbrace{adA(h^{2i+1})}_{RS1(c^i)})$

Klausur: $sxtImm(h^{2i+1}) = sxtImm(c^i)$ *

$lop(h^{2i+1}) = lop(c^i)$

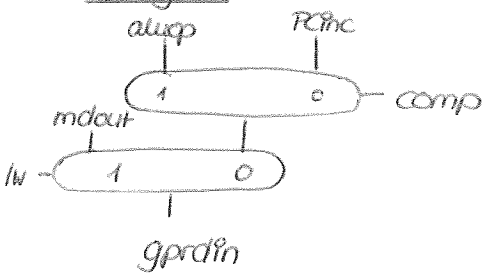
$rop(h^{2i+1}) = \begin{cases} B(h^{2i+1}) : rtype(h^{2i+1}) \\ sxtImm(h^{2i+1}) : sonst \end{cases} \overset{*}{=} \begin{cases} c_i.gpr(RS2(c_i)) : rtype(c_i) \\ sxtImm(c^i) : sonst \end{cases} = rop(c^i)$

$aluf(h^{2i+1}) = aluf(c^i)$ HW-Fu., Instruction decoder

$aluop(h^{2i+1}) = aluop(lop_h(h^{2i+1}), rop_h(h^{2i+1}), aluf_h(h^{2i+1}))$

$= aluop(lop(c^i), rop(c^i), aluf(c^i))$

GPR glue



$$gprdin(h^{2i+1}) = \begin{cases} alucp(h^{2i+1}) : comp(h^{2i+1}) \\ PCinc(h^{2i+1}) : link_4(h^{2i+1}) \\ mdout(h^{2i+1}) : lw(c^i) \end{cases}$$

$$= \begin{cases} alucp(c^i) : comp(c^i) \\ c^i.pc +_{32} 4_{32} : link(c^i) & \text{später ①} \\ c^i.m_4(ea(c^i)) : lw(c^i) & \text{später ②} \end{cases}$$

$$gprw = ex \wedge (comp \vee link)$$

$$comp(c^i) \Rightarrow gprw(h^{2i+1}) = 1$$

$$h^{2i+1}.gpr(x) = \begin{cases} gprdin(h^{2i+1}) : x = adC(h^{2i+1}), x \neq 0^5 \\ h^{2i+1}.gpr(x) : sonst \end{cases}$$

$$= \begin{cases} alucp(lop(c^i), rop(c^i), aluf(c^i) : x = RD(c^i), x \neq 0^5 \\ c^i.gpr(x) \end{cases}$$

$c^i.m : \{0,1\}^{32} \to \{0,1\}^8$ Byte-Adressiert

$h^t.m : \{0,1\}^{30} \to \{0,1\}^{32}$ Wort-Adressiert

24.05.2006

$$\boxed{h^t.ex = t \bmod 2}$$

Satz: $\left.\begin{array}{l} h^{2i}.pc = c^i.pc \\ h^{2i}.gpr = c^i.gpr \\ x \in \{0,1\}^{30} : h^{2i}.m(x) = c^i.m(x00) \end{array}\right\} \forall i$

fetch: t gerade $h^2.ex = 0$

$$h^{2i+1}.IR = I(c^i)$$

execute: $p_h(h^{2i+1}) = p(c^i)$

$$f_h(h^{2i+1}) = f(c^i)$$

$$h^{2i+2}.gpr = c^{i+1}.gpr$$

e.e. 1) $nextpc(h^{2i+1}) = c^i.pc +_{32} 4_{32}$ link

2) $mdout(h^{2i+1}) = c^i.m_4(ea(c^i))$ lw

Graphik aus Skript 2006, S. 18

ad 1) $A_h(h^{2i+1}) = A(c^i) = c^i.gpr(RS(c^i))$

$jbtaken_h(h^{2i+1}) = jbtaken(c^i)$

$\Rightarrow Aeqe(c^i) = (A_h(h^{2i+1}) = 0^{32})$

$h^{2i+2}.pc = c^{i+1}.pc$

$pcinc(h^{2i}) = pcin(h^{2i+1})$

$= c^i.pc +_{32} 4_{32} = (c^i.pc[1:0] = 00)$ qed 1)

Änderung zur Skriptgraphik: Signal pcinc nach 30-Bit Inkrementer abgreifen!

ad 2) $mW_h = SW_h \wedge ex$

$\Rightarrow mW_h(h^{2i}) = 0$

$\Rightarrow h^{2i+1}.m = h^{2i}.m$

$mad(h^{2i+1}) = c^i.gpr(RS1(c^i)) + sKtimm(c^i)$

$mdout(h^{2i+1}) = h^{2i+1}.m(\underbrace{mad(h^{2i+1})}_{x})$

$= c^i.m_4(mad(h^{2i+1})00)$

$= c^i.m(ea(c^i))$

$SW(c^i) \Rightarrow B(h^{2i+1}) = \underbrace{h^{2i+1}.gpr(h^{2i+1}.IR[20:16])}_{adB(h^{2i+1})}$

$= \underbrace{h^{2i}.gpr}_{c^i.gpr}(\underbrace{I(c^i)[20:16]}_{RD(c^i)})$

$h^{2i+2}.m(\underbrace{mad(h^{2i+1})}_{ea[31:2]}) = c^i.gpr(RD(c^i))$

$= c^{i+1}.m_4(ea(c^i))$ qed 2)

Ausblick: CO-Compiler

(Pascal mit C Syntax)

- Kontextfreie Grammatiken
- CO-Syntax
- CO-Semantik $\delta c(c) = c'$ Konfig. der abstrakten CO-Maschine
- Compiler: CO → DLX (Simulationssatz)

Definition: Kontextfreie Grammatiken (context free grammars CFG)

$G = (T, N, S, P)$

T: Menge der Terminale
N: Menge der Nichtterminale

$S \in N$ Startsymbol
$P \subset N \times (N \cup T)^*$ Produktionsregeln

Exkurs: Menge A

Menge der Wörter in A der Länge n: $A^n = \{(a_1 \dots a_n) \mid \forall i : a_i \in A\}$

Menge aller Wörter aus A: $A^* = \bigcup_{i=0}^{\infty} A^i$, $A^0 = \{\varepsilon\}$ leeres Wort

$p \in P$: $p = (n, w)$ $\quad n \in N, w \in N \cup T$

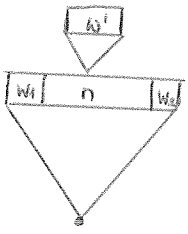
Für $(n, w) \in P$ schreiben wir auch: $n \to w$ (w ist direkt ableitbar von n)

$(n, w_1) \in P \wedge \dots \wedge (n, w_k) \in P$: $n \to w_1 \mid w_2 \mid \dots \mid w_k \in (N \cup T)^*$

Sprache von G: $\mathcal{L}(G) = \{w \mid w \in T^* . S \xrightarrow[G]{*} w\}$

Definition: Ableitungsbäume von G

- S ist ein Ableitungsbaum mit Wurzel S und Blattwort S
- Sei Q Ableitungsbaum mit Wurzel n und Blattwort $w = w_1 n w_2$
  $n \to w' \in P$, dann ist Q' Ableitungsbaum mit Wurzel n
  und Blattwort $w = w_1 w' w_2$



Definition: $n \xrightarrow[G]{*} w$ w kann aus n mit Grammatik G in $*$ Schritten abgeleitet werden $\iff \exists$ Ableitung mit Wurzel n und Blattwort w

alternativ: $u, u' \in (N \cup T)^*$

$u \xrightarrow{}_G u' \iff \exists w_1, w_2, n$ und $n \to \omega \in P$: $u = w_1 n w_2$, $u' = w_1 \omega w_2$

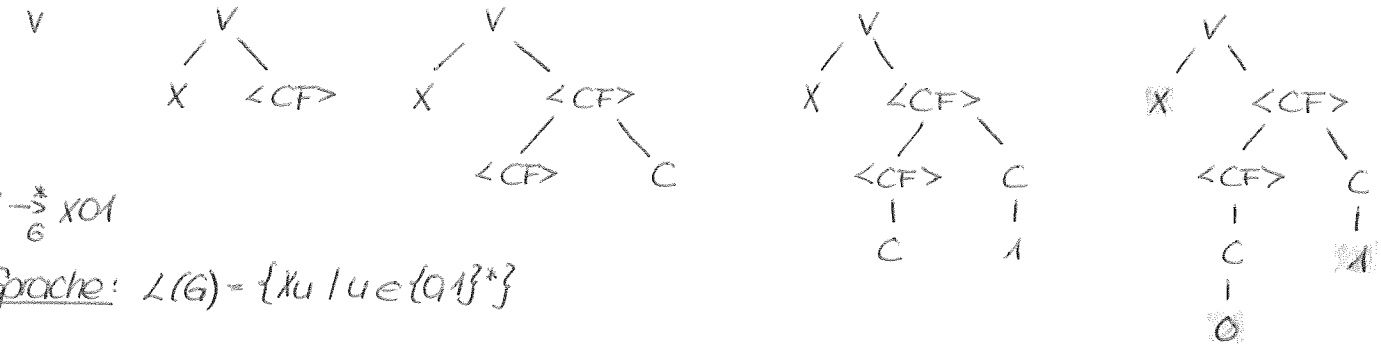
$u \xrightarrow[G]{*} u' \iff \exists v_0, v_1, \dots v_s : v_s = v'$ $v_i \to v_{i+1}$ $\forall i$ (Ableitungen)

$\mathcal{L}(G) = \{v \mid S \xrightarrow[G]{*} v\}$

Beispiel: $N = \{V, C, \langle CF \rangle\}$, $T = \{X, 0, 1\}$, $S = V$

P: $C \to 0 \mid 1$ Konstanten
$V \to X \mid X\langle CF \rangle$ Variablen
$\langle CF \rangle \to C \mid \langle CF \rangle C$ Konstantenfolgen

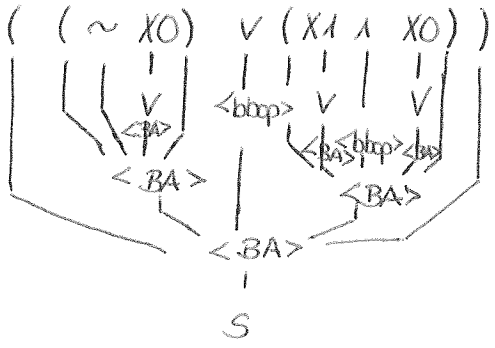
Herleitung von X01



$V \overset{*}{\underset{G}{\to}} X01$

Sprache: $L(G) = \{Xu \mid u \in \{0,1\}^*\}$

Erweiterung der Grammatik 1:



$V \neq S$

$S \to \langle BA \rangle$ Boolescher Ausdruck

$\langle BA \rangle \to C \mid V \mid (\sim \langle BA \rangle) \mid \langle BA \rangle \langle bbop \rangle \langle BA \rangle$

$\langle bbop \rangle \to \wedge, \vee, \oplus$

Erweiterung der Grammatik 2:

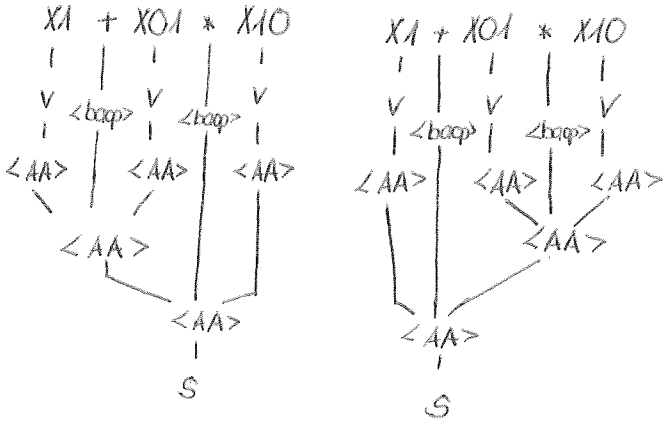
$\langle AA \rangle$ Arithmetischer Ausdruck

$V \to$, $C \to$ } wie oben

$\langle AA \rangle \to C \mid V \mid -_A \langle AA \rangle \mid (\langle AA \rangle) \mid$

$\langle AA \rangle \langle baop \rangle \langle AA \rangle$

$\langle baop \rangle \to + \mid -_a \mid \cdot \mid /$



Ableitung von $(X1 + X01 * X10)$ ist nicht eindeutig! $\Rightarrow$ Uneindeutige Grammatik!

Definition: Die Grammatik G heißt eindeutig $\Leftrightarrow$

$\forall w \in L(G)$ $\exists!$ Ableitungsbaum Q zu G mit Wurzel S

und Blattwort w

$\exists!$ existiert genau 1, Paulsche Vorstehen $\frac{1}{\exists}$

$\langle VC \rangle \to V \mid C$

$\langle F \rangle \to \langle VC \rangle \mid -_1 \langle F \rangle \mid (A)$ Faktor

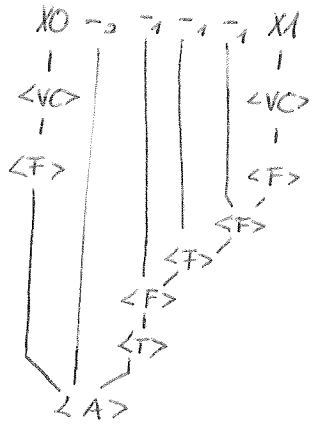
$\langle T \rangle \to \langle F \rangle \mid \langle T \rangle * \langle F \rangle \mid \langle T \rangle / \langle F \rangle$ Terme

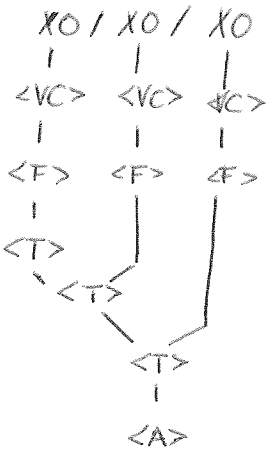
$\langle A \rangle \to \langle T \rangle \mid \langle A \rangle + \langle T \rangle \mid \langle A \rangle -_2 \langle T \rangle$ Ausdrücke

hoch — Priorität — niedrig

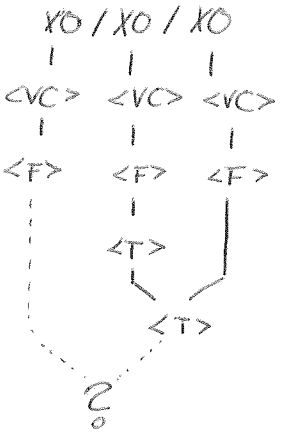
Satz: Diese Grammatik ist eindeutig.

Beweis: Loeckx, Mehlhorn, Wilhelm





Linksassoziativ !



Parsen: Bildung eines Ableitungsbaums zu gegebenem Blattwort

Ausdrucksgrammatik

31.05.06

$\langle At \rangle \to \langle A \rangle \langle Rel \rangle \langle A \rangle \mid 0 \mid 1 \mid V$ Atom

$\langle Rel \rangle \to = \mid \neq \mid < \mid \leq \mid > \mid \geq$ Relation

$\langle BF \rangle \to \langle At \rangle \mid \neg \langle BF \rangle \mid (\langle BA \rangle)$ Boolescher Termel/faktor

$\langle BT \rangle \to \langle BF \rangle \mid \langle BT \rangle \wedge \langle BF \rangle$ Boolescher Term

$\langle BA \rangle \to \langle BT \rangle \mid \langle BA \rangle \vee \langle BT \rangle$ Boolescher Ausdruck

Anweisungen:

$\langle An \rangle \to \langle Id \rangle = \langle A \rangle \mid \langle Id \rangle = \langle BA \rangle \mid$ Zuweisungen

while $\langle BA \rangle$ do $\{ \langle Anf \rangle \}$ | Schleife

if $\langle BA \rangle$ then $\{ \langle Anf \rangle \}$ else $\{ \langle Anf \rangle \} \mid$ } Verzweigung

if $\langle BA \rangle$ then $\{ \langle Anf \rangle \} \mid$ } Verzweigung

$\langle Na \rangle (\langle PF \rangle)$ Funktionsaufruf

$\langle Na \rangle = new \langle Typ \rangle$ Speicher anfordern

<PF> → <A> | <PF>, <A> | ε Parameterfolge
<AnF> → <An> | <AnF>; <An> Anweisungsfolge

Alternative Definition der Verzweigung:

<An> → if <BA> then <AnF> else <AnF> |
if <BA> then <AnF>

if BA then if BA then An else An
{ { } } Mehrdeutigkeit der Interpretation ↯
⇒ Läßt sich durch Klammerung der <AnF> auflösen!

Grammatik für CO-Programm

<program>	→ <DF>; <AnF>		Programm
<DF>	→ <D> | <D>; <DF>		Deklarationsfolge
<D>	→ <VaD> | <FuD> | <TypD>		Deklaration
<VaD>	→ <Typ> <Na>		Variablen-Deklaration
<Typ>	→ <elTyp> | <elTyp> [<ZeiF>]		el. Typen, Array
	<elTyp> * |		Pointer
	<Na> | <Na> [<ZeiF>] | <Na> * |		selbstdef. Typen
	struct { <KompDF> }		structs
<TypD>	→ typedef <Typ> <Na>		Typdeklaration
<elTyp>	→ int | bool | char | float | double | uint		elementare Typen
<ZeiF>	→ <Zei> | <Zei> <ZeiF>	<Zei> → 0 | ... | 9	Ziffern / Ziffernfolge
<Na>	→ <Bu> | <Bu> <BuZeiF>	<Bu> → a | b | ... | z | A | ... | Z <BuZei> → <Bu> | <Zei>	<BuZeiF> → <BuZei> | <BuZei> <BuZeiF>
<KompDF>	→ <KompD> | <KompD>, <KompDF>		Komponentendekl. / -folge
<KompD>	→ <Na> : <elTyp> | <Na> : <Na>		
<VaDF>	→ <VaD> | <VaD>; <VaDF>		Var. dekl. folge

Beispiele:
- Array Typen

int [15]

x7 [3] // vorherige Typdeklaration von x7

Mehrdimensionales Array

```
typedef int [9]   SP ;  // Spalte
typedef SP[9]   MA ;  // Matrix(9×9)
MA P ;    // Elemente P[i][j]   i,j∈{0..8}
```

Struct Typen

```
typedef LEL * u ;  //Listenelement
typedef struct {content: int, next: u} LEL ;
```

Kontextbedingungen in Typdeklarationen

1) typdef X [EIF> I:X muß vorher definiert sein
2) typedef {n1:: x1, ..., ns:: xs} Y : x1... xs muß vorher definiert sein
3) typedef X* Y : X darf nachher definiert sein

Funktionsdeklaration <FuD> : im Skript n

<Id> → <Na> | <EIF> | <Id> <Namen> | Name, Elfernfolge, Structurkomponente
<Id> [<A>] | Arrayzugriff
<Id> * | Dereferenzieren
& <Id> Address of

CO : Ausdrücke

07.06.2006

Identifier PCB[17]. GPR[24]
AX *. Q*

Name S S: Selektor S = S1S2 ...
Si [e] e: Ausdruck
. n n: Name
* dereferenzieren / Folge: Pointer

Typen voll rekursive Typen
& address of

Statements: if-then-else, while, assignments, (funct.) call, return,
new

Definition: Abstrakte CO-Maschine

K_p : Konfiguration zu Programm p

Parameter: deklarierte Typen, Variablen, Funktionen, die sich während der Laufzeit des Programms nicht ändern

$\delta_c : K_p \to K_p$

$\delta_c(c) = c'$ Ausführung eines Statements

$c \in K_p$; $c = \{c.pr, c.rd, c.lms, c.hm, c.rbs\}$

- c.pr : Programmtext, Anf
- c.rd : recursion depth : # geschachtelter noch nicht terminierter Funktionsaufrufe
- c.lms : local memory stack

  $c.lms[0 : c.rd] \to \{m \mid m \text{ Memory}\}$

  c.lms(0) global memory (globale Variablen)
- c.hm : heap memory : enthält mit new erzeugte, namenlose Variablen
- c.rbs : return binding stack : Subvariable, an die Return Wert übergeben wird

  $c.rbs : [0 : c.rd] \to \{s \mid s \text{ Subvariable}\}$

Statische Informationen in p

$Tname_p = \{t \mid t \text{ elementarer Typ}\} \cup \{t \mid \text{name eines decl. Typs}\}$

Definiere Ordnung auf Tname :

$t < t'$ t ist vor t' deklariert

$(p = \text{typedef } t, \ldots \text{ typedef } t')$ oder t elementar, t' elementar

$Typetable_p$: $Tname \to \{td \mid td \text{ type descriptor}\}$

te : elementarer Typ : $\#(te) = te$

$t = t'[n]$ mit $t' < t$

$\#(t) = t'[n]$

t = struct $\{n_1 : t_1, \ldots n_k : t_k\}$ $\quad t_1 .. t_k < t$

$tt(t)$ = struct $\{n_1 : t_1, \ldots n_k : t_k\}$

$t = t' *$ $\quad t' \in$ Tname

Allgemein: typedef n typedescriptor tt(n) = typedescriptor

Beispiel: typedef n LEL *

typedef LEL struct {cnt : int, next n}

tt	t	tt(t)
	int	int
	⋮	
	n	LEL *
	LEL	struct {cnt : int, next n}
	x	int [5]
	y	x [5]

t einfach, falls t elementar oder $t = t' *$

Definition: size(t): # einfachen Komponenten/Subkomponenten von t

t einfach : size(t) = 1 ⚠ t' nicht benutzt

$tt(t) = t'[n]$: size(t) = n · size(t')

$tt(t) =$ struct $\{n_1 : t_1 .. n_k : t_k\}$: $size(t) = \sum_{i=1}^{k} size(t_i)$ $\quad t_i < t$

Beispiel: $f(n) = f(n-1)$ $\quad n \in \mathbb{Z}$

$f(0) = f(-1) = f(-2) \ldots$ unendliche abst. Ketten

Aber: Table ist endlich

Allgemeinste Form: rekursive Definitionen, Induktionsbeweise } mit allg. wohlgeordneten Mengen

Theorie: Cantor'sche Mengenlehre, Ordinalzahlen

```
int up (int x)
{ int y ;
  if x=0 {y=0}
     else {y= up (x+1)};
  return y}
```

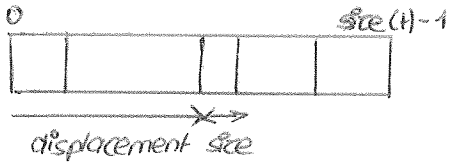
Beispiel zur Wohlordnung

$t \in Tname$

$Ra(t)$: Menge der Werte, die Variablen vom Typ t annehmen dürfen (Range)

$Ra(t) = \{ f \mid f[0 : size(t)-1] \to \{w \mid w \text{ einfacher Wert}\}$

$f \in Ra(int) : f : \{0\} \to T_{32} = \{-2^{31} : 2^{31}-1\}$



1) $t = t'[n]$

2) $t = struct\{n_1 : t_1 \dots n_k : t_k\}$

Displacement:

$$displ(i,t) = i \cdot size(t') \quad i<n$$

$$displ(n_i,t) = \sum_{1 \le j < i} size(t_j)$$

~~zu 1) $f : \{size(t')\} \to displ$~~

zu 1) $f_{size(t')}(displ(i,t)) \in Ra(t)$

zu 2) $f_{size(t_i)}(displ(n_i,t)) \in Ra(t_i)$

3) offen: $t = t'*$

$f \in Ra(t) \quad f(0) = Variable$

↑ künstliche Folge der Länge 1

einfach: elementar (int, char), pointer

Erinnerung: $c = (c.pc, c.rd, c.lms, c.hm, c.rbs)$

$c.lms\ [0 : c.m] \to \{m \mid m \text{ Memory}\}$

$c.hm \in \{m \mid m \text{ Memory}\}$

Variable (m,i) m Memory

$i \in \mathbb{N}_0$, i-te Variable in Memory m

Memory m:

$m = (m.n,\ m.name,\ m.typ,\ m.ct)$

$m.n \in \mathbb{N}_0$ — Anzahl der Variablen

$m.name : [0 : m.n-1] \to \{n \mid n \text{ Name}\}$ injektiv — Name der i. Variable

$m.typ : [0 : m.n-1] \to Tname$ — Typ der i. Variable

$m.ct : [0 : size(m)-1] \to \{w \mid w \text{ einfacher Wert}\}$ — Inhalt der i. Variable

$$size(m) = \sum_{i=0}^{m.n-1} size(m.typ(i))$$

Subvariablen in m:

- (m,i) alle Variablen sind Subvariablen
  - $(m,i) \in$ Subvariable vom Typ $m.typ(i)$
- $(m,i)S$ Subvariable vom Typ $t = t'[n]$
  - $\Rightarrow j < n : (m,i)S[j]$ Subvariable vom Typ t'
- $(m,i)S$ Subvariable vom Typ $t = struct\{n_1 : t_1, \dots n_k : t_k\}$
  - $\Rightarrow \forall j < k : (m,i)S.n_j$ sind vom Typ t_j

Werte von Pointern: Subvariablen

Wo stehen Variablen und Subvariablen in m?

Basisadressen:

- $ba(m,i) = \sum_{j<i} size(m.typ(j))$
- $u = (m,i)S \qquad typ(u) = t'[n] = t$

  $ba((m,i)S[j]) = (ba((m,i)S) + displ(j, t')$
- $u = (m,i)S \qquad typ(u) = t = struct\{\dots\}$

  $ba((m,i)S.n_j) = ba((m,i)S) + displ(n_j, t)$

$m.ct[0 : size(m)-1] \to \{w \mid w \text{ einfach}\}$

$\forall$ Variablen $(m,i) : w.ct_{size(m,i)}(ba((m,i)S) \in Ra$

$size(m,i) = size(m.typ(i))$

Werte einer Subvariablen $(m,\pi)S$ in Konfig c

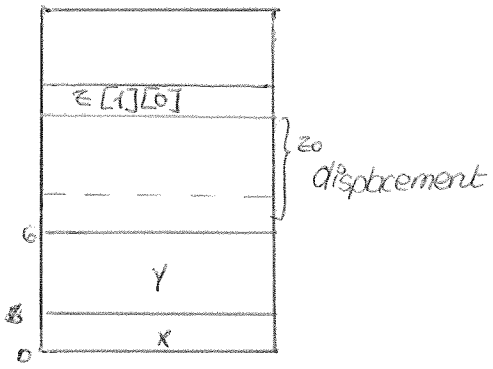
$va\ (c, (m,\pi)S) = m.ct_{size(typ((m,\pi)S))}\ ba((m,\pi)S)$

Beispiel: `typedef int [5] row;`

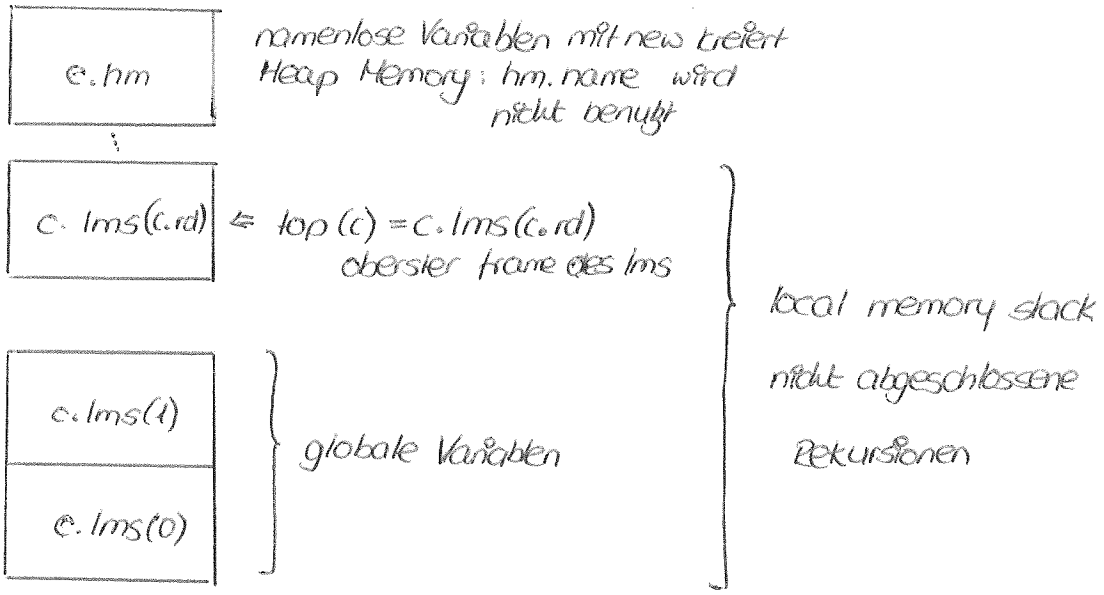
`typedef row [5] matrix;`

`int x;`
`row y;`
`matrix z;`

$size = 1 + 5 + 25 = 31$



Memories der C-Konfiguration C



lms (?) müssen zum Programm passen

lms(0) $Gname = \{v \mid v$ Name globaler Variablen$\}$

lms(0) $n = \#Gname$

size. (lms(0), ?) ?-te globale Variable in Reihenfolge d. Deklarationen

lms(0).name(?)
.typ(?)

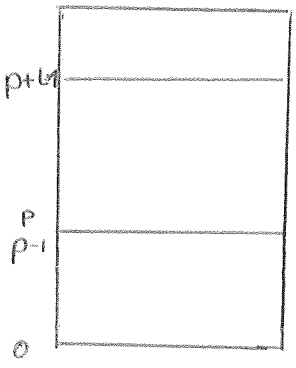
Später bei Definition der Semantik von Funktionsaufrufen

Aufruf $rd = f(x_0, \dots x_p)$ f int, p Parameter
lokale Variablen

$c'.rd = c.rd + 1$

$top(c')$ = Oberster neuer Frame

$top(c').n = p+L$



$top(c')$ ist mit aufrufenden Funktion f kompatibel.

$$top(c').name(i) = \begin{cases} \text{Name des } i\text{. Parameters} & i < p \\ \text{Name der } (i-p)\text{. lok. Var.} & i \geq p \end{cases}$$

$typ(i)$ genauso!

14.06.2006

$(m,i)\,S$ Subvariablen

$S = S_1 \dots S_k$ Selektoren

$S_i[x]$ $x \in \mathbb{N}_0$ Arrayelement selektieren
.n .struct

$ba((m,i)\,S[x]) = ba((m,i)\,S) + displ(x, typ(m,i)S)$

$ba((m,i)\,S.n_j) \dots\dots + displ(j, typ(m,i)S)$

Wert einer Subvariablen $(m,i)S$ in einer Konfiguration c

$$va(c, (m,i)S) = mem_{size(typ(m,i)S))}(ba((m,i)S))$$

Ausdrucksauswertung

e Ausdruck &e 'Adress of' e (Adresse, an der Wert von e steht)

- nicht möglich: &1
  &&e
- falls möglich: $va(op) = va(c, \&e)$

Induktive Definition von über Aufbau von Ausdrücken

\ / Abbildungs
e

Def. von Blättern zur Wurzel
"<" Abstand zur Wurzel

- $e \in [0,1]^*$ Konstante

  & e nicht definiert

  $va(c,e) = [e]$ Integer

  $va(c,e) = e$ boolean

- $e = X$ (Variablen) Name

  $$\&(c,X) = \begin{cases} (top(c), j) & : \exists j : top(c).name(j) = X \quad (name\ injektiv) \\ (lms(0), j) & : \exists j : \lfloor lms(0).name(j) = X \\ \text{Fehler} & \text{sonst} \end{cases}$$

  binde Namen X an Variable

- arrays: $e = e'[e'']$

  $$\&(c,e) = \underbrace{(\&(c,e'))}_{(m,l)s}\ [va(c,e'')]$$
  neuer Selektor s

- structs: $e = e'.n$

  $$\&(c,e) = (\&(c,e)).n$$

- Ausdrücke: $e = e' + e''$

  & e nicht definiert

  $$va(c,e) = va(c,e') + va(c,e'')$$

- Pointer dereferenzieren: $e = e'*$ $\quad typ(e') = t*$

  $$\&(c,e) = va(c,e')$$

- Adress of: $e = \&e'$

  & e nicht definiert

  $$va(c,e) = \&(c,e')$$

Beispiel: $(\&e')*$

$\&(c_1, \underbrace{(\&e')*}_{e}) = va(c_1, \&e') = bin\&(c_1, e')$

$va(c_1, (\&e')*) = va(c_1, \&(c_1 e)) = va(c_1, \&(c_1 e')) = va(c_1, e')$

Anweisungen ausführen

C° Startkonfiguration

$c^\circ.rd = 0$ $\quad c^\circ.rbs(0)$ nicht definiert

program $d;a$ $\quad d$: Deklarationsfolge, a: Anweisungsfolge

$c^\circ.pr = a$

ab jetzt: $c^{i+1} = dc_0(c^i)$

$c' = dc_0(c)$

Fallunterscheidung in $a1$ in $c.pr = a1; r'$

- $a1$: if e then $\{a\}$ else $\{b\}$

$$c'.pr = \begin{cases} a; r' & : va(c,e) = 1 \\ b; r' & : va(c,e) = 0 \end{cases}$$

- $a1$: while e do $\{a\}$

$$c'.pr = \begin{cases} a; \text{while } e \text{ do } \{a\}; r' & m(c,e) = 1 \\ r' & m(c,e) = 0 \end{cases}$$

- Zuweisung: $a1$: $id = e; r'$

$va(c', id) = va(c, e)$

$c'.pr = r'$

$va(c', id) = va(c', \underbrace{\&(c', id)}_{\&(c, id)})$ Subvariable u

$u = (m, i) \in S$

$m.ct_{size(u)}(ba(u))$

Nacharbeiten bis S. 40 im Skript!

19. 06. 2006

Korrektheitsbeweise in CO-Semantik

① `int x;`
`x = 3;`
`if x==0 then {x=1}`
`else {x=2}`

Ziel: $\varepsilon\varepsilon$: Am Programmende gilt

$x = 2$

P-Regeln		Werte
$c^0.pr$	`x=3; if x==0 then {x=1}else{x=2}`	$va(c^0, x) = undef.$
$c^1.pr$	`if x==0 then {x=1} else {x=2}`	$va(c^1, x) = 3$
$c^2.pr$	`if` $x=2$	$va(c^2, x) = 2$
$c^3.pr$	ε	$va(c^3, x) = 2$

② `int n;`
`int result;`
`result = 0;`
`n = A;` A Konstante
`while n>0 do{`
`result = result + n;`
`n = n - 1;`
`}`

z.z. $\exists t.\ c^t.pr = \varepsilon$

$\wedge\ va(c^t, result) = \sum_{i=1}^{A} i \bmod 2^{32}$

Nach dem j. Durchlauf der Schleife:

c^{2+3j} : $va(c^{2+3j}, result) = \begin{cases} 0 & j=0 \\ A + \ldots + (A-j+1) & j>0 \end{cases}$

$va(c^{2+3j}, n) = A - j$.

Induktionsschritt $j \to j+1$:

$c.pr$ = `while e do {a}`

$c'.pr$ = `{a}; while e do {a}`

IS: 1. Durchlauf

IV: alle übrigen Durchläufe

Problem: Induktionsschritt über Durchlauf, aber while selbst über Durchlauf definiert.

Induktionsbehauptung:

Sei $va(c^\alpha, result) = K$

$va(c^\alpha, n) = M$

$\forall k, M \quad c^\alpha.pr = \text{while } n>0 \text{ do } \{\}$

$c^{\alpha+3M}$ weitere Durchläufe

$va(c^{\alpha+3M}, result) = K + M + (M-1) + \ldots + 1$ } Beweis Übung

$va(c^{\alpha+3M}, n) = 0$

Vereinfachung der Notation:

$c.pr = id = f(e_0, \ldots e_{p-1}); r'$

$\Rightarrow c'.pr =$ Rumpf von $f; r'$

Deklarationen: Tabellen für die Semantik! (Parameter ändern sich nicht im Verlauf des Programms)

z.B.: $tt : Tname \to fd$

Benötige ebenso Funktionstabelle: $ft : FKname \to fd$ (function descriptor)

$Fname = \{f \mid f \text{ deklariert. Fktsname } \vee f = main\}$

$fd = \{fd.typ, fd.body, fd.p, fd.n, fd.name, fd.typ\}$

$fd.typ \in Tname$ Name des Typs des Rückgabewerts

$tt(\underbrace{ft(f)}_{d}.typ)$ Descriptor des Typs des Rückgabewerts von f

$fd.body$: Rumpf f

$fd.p$: Parameteranzahl

$fd.n$, $fd.name$, $fd.typ$ } wie bei Memory

Anzahl der lokalen Parameter: $fd.n - fd.p$

Compiler CO → DLX

$p = d; a$ d: Deklarationsfolge, a: Anweisungsfolge

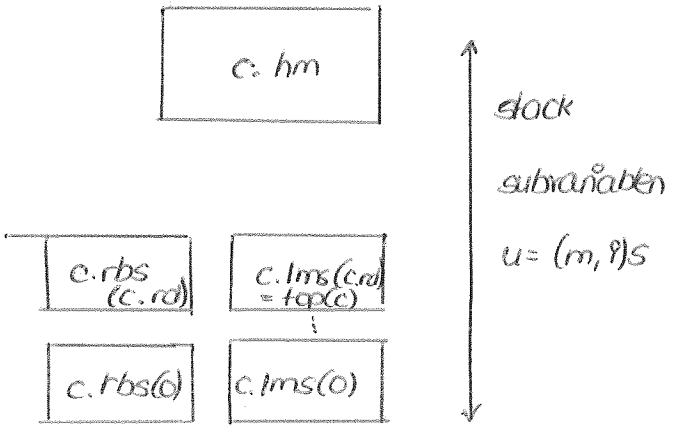
code (p): DLX Programm

Wald von Ableitungsbäumen

1) geringste Übersetzung der Ableitungsbäume (Rekursion über Bäume)

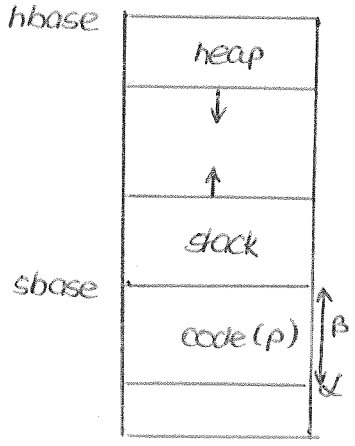
2) Sprungdistanzen einsetzen für Funktionsaufrufe

CO-Config c



DLX d

21.06.2006



allocated base address: aba $(m,i) \in [0,1]^{32}$

Wert einer Variablen: $d.m_{asize(m,i)s}\ (aba((m,i)s))$

asize: allocated size

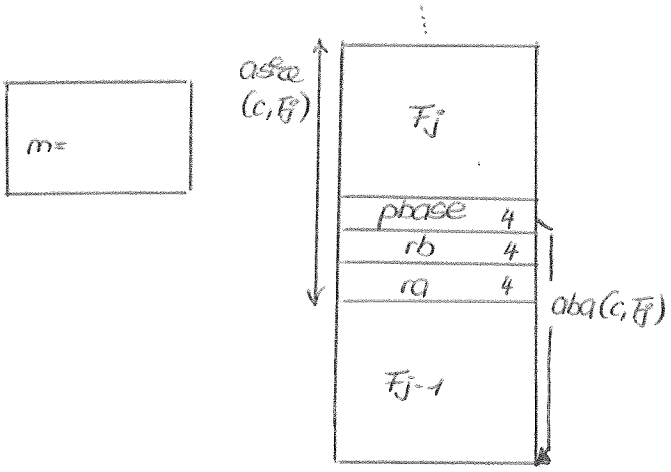
Zu einer Rechnung $c^0, c^1 \ldots$ der CO-Maschine gibt es
eine Rechnung $d^0, d^1 \ldots$ der DLX-Maschine,
eine Funktion $aba^0, aba^1 \ldots$ und eine Folge von Schritten $s(0), s(1) \ldots$,
so daß gilt:

Satz: $\forall t \quad consis(c^t, aba^t, d^{s(t)})$

Bemerkungen: • consis betrifft Werte von Variablen

• aba^t, aba^{t+1} unterscheiden sich höchstens durch den Definitionsbereich (solange keine Garbage Collection)

• Nach $S(t)$ DLX-Schritten sind t CO-Schritte simuliert



DLX Stack-frame

elementarer Wert: 4 Bytes

Pointer: 4 Bytes

pbase: baseaddress of previous stackframe

rb: speichern des Returnwerts ab dieser Adresse

ra: Return Code Address

$$asize(c, F_j) = size(c.lms(j)) \cdot 4 + 12$$

$$aba(c, F_j) = sbase + \sum_{i<j} asize(c, F_i)$$

$$aba(c, c.lms(j), p)s) = aba(c, F_j) + 12 + 4 \cdot ba(k, lms(j), 1).s)$$

Definition: $consis(c, aba, d) \leftrightarrow$

$e\text{-}consis$ — elementare Typen

$\wedge\ p\text{-}consis$ — Pointer

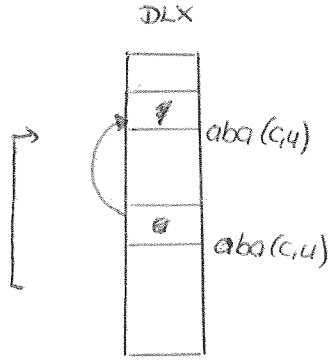
$\wedge\ r\text{-}consis$ — Rekursion (Stackverwaltung)

$\wedge\ c\text{-}consis$ — Control (PC-Anforderungen)

1) $e\text{-}consis(c, aba, d)$

$\forall\ u$ aus c mit $typ(u)$ elementar

$$va(c,u) = d.m_4(aba(c,u))$$



2) $p\text{-}consis$

$\forall u$ aus c mit $va(c,u) = y$ (Pointer)

$$d.m_4(aba(c,u)) = aba(c,y)$$

(mit Garbage Collection: falls u erreichbar: Graphensuche über Variablen auf Stack durch verfolgen von pointern)

3) L-consts

Setze 3 general purpose register dedicated ein.

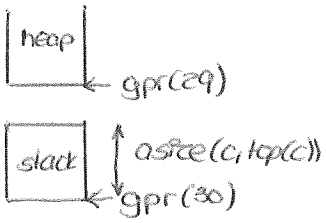
d.gpr(28) = sbase : global memory pointer

d.gpr(29) : heappointer

d.gpr(30) = aba(c, top(c)) : stack pointer

a) $d.gpr(30) + asize(c, top(c)) \leq d.gpr(29)$

heap und stack sind disjunkt



b) pbase von f_j

$\forall j \in [1..ara]$

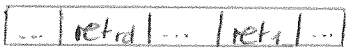
$$d.m_4(aba(c, c.lms(j))) + \delta = aba(c, c.lms(j-1))$$

pbase von f_j zeigt auf Basisadresse des vorigen stack frames

c) lb ⚠ c.lbs(j) Subvariablen

$$\underbrace{d.m_4(aba(c, c.lms(j)+4))}_{rb \text{ von } f_j} = aba(c, c.lbs(j))$$

d) c.pr : enthält rd returns



l_j : erste Anweisung nach ret_j im Programmtext

co_j : vom Compiler erzeugter Code für l_j

start(c, j) : Startadresse von c_j

$$\forall j \quad \underbrace{d.m_4(aba(c, c.lms(j)))}_{rb \text{ von } f_j} = start(c, j)$$

4) C-consts

head(c.pr) : erste Anweisung d. Programmtexts

code(head(c.pr)) : vom Compiler erzeugter Code

start(c) : Startadresse dieses Codes

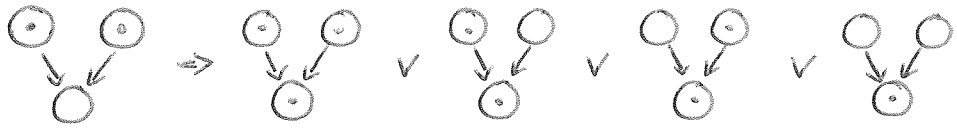
$$d.pc = start(c)$$

Züge:

1) Wenn die Quelle leer ist, dann darf eine Marke gesetzt werden (Eingabe)

2) Eine Marke darf immer weggeworfen werden
(nicht mehr benötigte Zwischenergebnisse)

3) Ein Nachfolger darf markiert werden, wenn alle Vorgängerknoten markiert sind (Rechnen)



Start: keine Marke auf Graph

Ziel: Auf jede Senke in irgendeinem Zeitpunkt Marke setzen

Baum: 1. Senke = Wurzel

Knoten: Problemgröße
Züge: Rechenzeit (auf Bäumen nicht interessant)
Marken: Speicherplatz

Aho - Ullmann Algorithmus

$T(n) = \min \{ x \mid$ jeder Baum mit Ingrad 2 und n inneren Knoten kann mit x Marken markiert werden $\}$

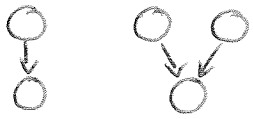
Satz: $T(n) \leq \lceil \log_2 n \rceil + 2$

$T(n) = 27$

$n = 2^{25}$

Beweis: Angabe der Markierungsstrategie

Induktion:

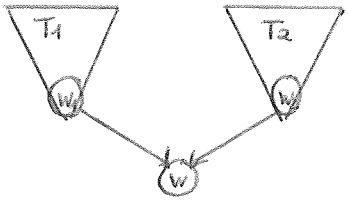
Induktionsanfang: $n = 1$ 

Induktionsschluß: $n \rightsquigarrow n+1$:

Fall 1: Wurzel hat Ingrad 1

Beweis: Übung

Fall 2: Wurzel hat Ingrad 2



Beweis Fall 2: # inneren Knoten $T_1 \geq$ # inneren Knoten T_2

$\Rightarrow$ # inneren Knoten von $T_2 \leq \frac{n}{2}$

- Markiere w_1 mit Strategie T_1
- entferne alle Marken auf T_1 außer Marke auf w_1
- Markiere w_2 ....
- entferne alle Marken auf T_2 außer Marke auf w_2
- schiebe Marke von w_2 auf w

Marken: $T(n) \leq \max\{T(n-1), 1+T(\frac{n}{2}), 2\} \overset{!}{\leq} \lceil \log_2 n \rceil + 2$

$2 \leq \ldots + 2$ ✓

$T(n-1) \leq \lceil \log_2(n-1) \rceil + 2$

$\leq \lceil \log_2(n) \rceil + 2$ ✓

$1 + T(\frac{n}{2}) \leq 1 + \lceil \log_2(\frac{n}{2}) \rceil + 2$

$\leq 1 + \lceil \log_2(n) \rceil - 1 + 2 = \lceil \log_2(n) \rceil + 2$ ✓

Ausdrucksauswertung:

Bsp: $x = e1 + e2$ ← R-value: $va(c, e1+e2)$ ausgerechneter Wert

↑ CO Semantik
$\delta(c, x)$
wohin soll gespeichert werden

L-value: generierter Code aba (c, x)

$\forall$ Ausdrucksbäume T

$\forall$ Knoten n in T: $R(u) \in \{0,1\}$ berechnen

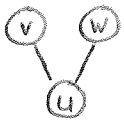
$$R(u) = \begin{cases} 1 : & R\text{-Value} \\ 0 : & L\text{-Value} \end{cases}$$

Definition: gehennt für Wurzel und Blätter

u Wurzel eines Ausdrucksbaums T

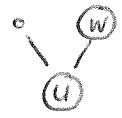
$$R(u) = \begin{cases} 0 : & T \text{ linke Seite einer Zuweisung} \\ 1 : & \text{sonst (rechte Seite Zuweisung, if-then-else, while, Param.)} \end{cases}$$

nicht Wurzel



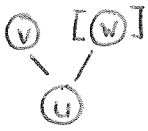
$u = v \circ w$ $\quad R(v) = R(w) = 1$

$\circ \in \{+, -, <, = \ldots\}$



$u = \circ w$ $\quad R(w) = 1$

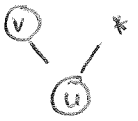
$\circ \in \{-, \neg\}$



$u = v[w]$ $\quad R(w) = 1$, $R(v) = 0$

$u = v.n_i$ $\quad R(v) = 0$

n_i: Struct Komponente



$u = v*$ $\quad R(v) = 1$ $\quad \triangle$

$u = \& v$ $\quad R(v) = 0$

Codeerzeugung für Ausdrücke e_i, Teilausdrücke in einer Funktion f

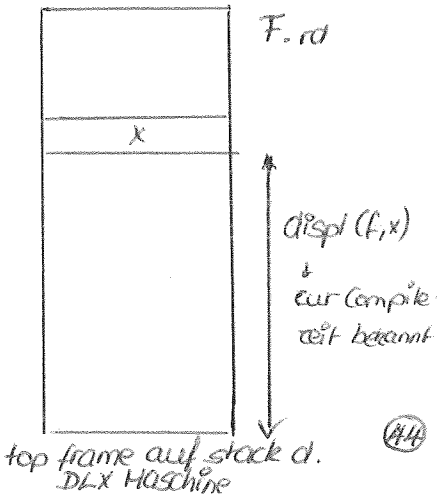
CO-Konfiguration c insbes. $top(c)$ konsistent mit T

DLX-Konfiguration d

$consis(c, aba, d)$

$$aba(c, x) = aba(c, m) + \overbrace{12 + 4 \cdot ba(m, i)}^{displ(f,x)}$$

$(m, i) = e(c, x)$ $\quad m = \begin{cases} top(c) : & x \text{ Ideal, Param. } m.name(i) = x \\ ins(0) : & \text{sonst} \end{cases}$



aba(c,x) = aba(c,m) + displ(f,x)

↳ stackpointer gpr 30

↳ sbase gpr 28

Codeerzeugung für Zugriff auf stack-Variablen

- Code für Variablenname X

1) aba(c,x) berechnen

in gpr(j) mit Markierungsstrategie markiert Blatt x mit Marke j

(wir benutzen Marken mit Nummern 1, 2, ... 27)

Fall: X lokal

$$gpr(j) = \underbrace{aba(c, top(c))}_{gpr(30)} + displ(f,x)$$

Code: addi RD = j, RS1 = 30, imm = displ(f,x)

Fall: X global

$$gpr(j) = \underbrace{sbase}_{gpr(28} + displ(f,x)$$

Code: addi RD = j, RS1 = 28, imm = displ(f,x)

Fallunterscheidung über Vorkommen von x:

R(x) = 0 : nach Ausführung des erzeugten Codes:

DLX-Konfiguration d': $d'.gpr(j) = aba(c, \underbrace{s(c,x)}_{(m,i)})$

R(x) = 1: typ(c,x) einfach

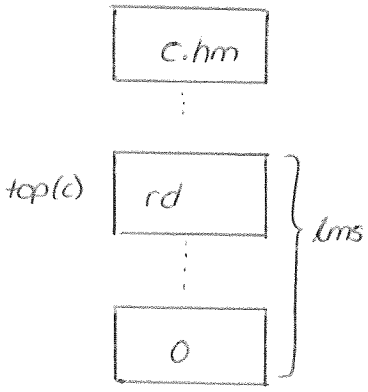
dereferenzieren

lw RD = j, RS1 = j, imm = 0

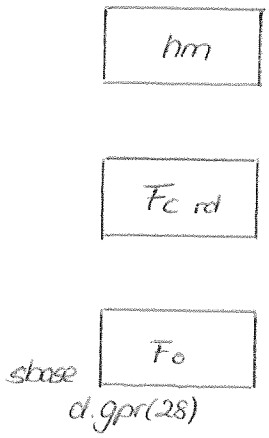
danach Konf d''

$$d''.gpr(j) = \begin{cases} va(c,x) & type(x) \text{ kein ptr.} \leftarrow econsts \\ aba(c, s(c,x)) & type(x) = pointer \leftarrow p\,consts \end{cases}$$

Wiederholung: 0-Conf c



DLX-Conf d



cons (c, aba, d)

e-consts : va (c,u) = d.m$_{size(typ(u))}$ aba (c,u)

p-consts : aba (va(cu)) = d. m$_4$ (aba (c,u))

L - consts

&(c,e) e = Var. name Bindung von Namen an Variablen

&(c,x) = (m,i)

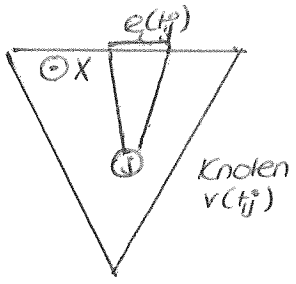
C-Sichtbarkeitsregeln: $m = \begin{cases} top(c) : X \text{ Lokal, Param.} \\ c.lms(0) : X \text{ global} \end{cases}$ m.name (i) = X

aba (c,x) = aba (c,m) + $\underbrace{4 \cdot (3 + ba(m,i))}_{displ(i,x)}$

aba (c,m) = $\begin{cases} d.gpr(30) : X \text{ Lokal} \\ d.gpr(28) : X \text{ global} \end{cases}$ ⇐ r-consts

X in body von f

Ausdrucksauswertung



e ... Baum ... Markenspiel

Eäge 1...T Marken haben Nummern ∈ {1..27}

Eug t ∈ {1..T} ⇝ ecode (t) : Expressioncode für Eug t

Eug t markiert Blatt X mit Marke j

Code(t) : addi RD = j RS1 = $\begin{cases} 30 : X \text{ lokal} \\ 28 : X \text{ global} \end{cases}$ imm = displ (i,x)

R = 1. deref

deref : lw RD = j RS1 = j imm = 0

Korrektheit der Ausdrucksübersetzung

Voraussetzung: consis (c, aba, d)

Sei $e(t,j)$ Teilausdruck des Knotens mit Marke j nach Zug t

Definition: Übersetzung der ersten t Züge

$$Ecode(t) = ecode(1) \dots ecode(t)$$

Behauptung: $\forall t \in \{1 \dots T\}$ — $d^{(t)}$ Ergebnis nach Ausführung von

$d^{(t)} : d \xrightarrow{*}_{Ecode(t)} d^{(t)}$ — $Ecode(t)$ beginnend in Konfig. d

$M(t) : \{j \mid$ Register j hat Marke nach Zug $t\}$

$\forall j \in M(t)$

$$d^{(t)}.gpr(j) = \begin{cases} aba(c, \&(c, e(t,j))) & : R(v(t,j)) = 0 \\ va(c, e(t,j)) & : R(v(t,j)) = 1 \wedge \neg \text{pointer} \\ aba(va(c, e(t,j))) & : \text{pointer} \end{cases}$$

Übersetzungen von Konstanten

$u = \tilde{c}$ — $\tilde{c}$: Konstante

$\tilde{c} = \{0, \dots 9\}^*$ — c: Konfigurationen

$\langle \tilde{c} \rangle_{10} \leq 2^{31} - 1$

Sei $\langle b \rangle_2 = \langle \tilde{c} \rangle_{10}$ Rechne Dezimalzahl in Binärzahl um

Ziel: $d^{(t)}.gpr(j) = b = va(c, u)$ — sign extension

lghi RD = j imm = $b[31:16] \oplus b[15]$

xori RS1 = j RD = j imm = $b[16:0]$

$d'.gpr(j) = b[31:16] \oplus b[15]^{16}, 0^{16} = b[31:16] \oplus b[15]^{16}, 0^{16}$

$d''.gpr(j) = d'.gpr(j) \oplus sxtimm(d')$

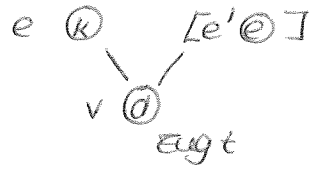
$= (b[31:16] \oplus b[15]^{16}, 0^{16}) \oplus b[15]^{16}\, b[15:0]$

$= (b[31:16] \oplus b[15]^{16} \oplus b[15]^{16}, 0^{16}) \oplus b[15:0]$

$= b[31:16], b[15]^{0}$

zeigt, daß obiger Code wirklich Konstante hochlädt

$v = e\,[e']$



$R(e') = 1$

$d^{(i-1)}.gpr(e) = val(c, e')$

$d^{(i-1)}.gpr(k) = aba(c, s(c,e))$

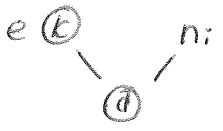
$type(e) = t'[u]$

$d^{(i)}.gpr(j) :\stackrel{!}{=} d^{(i)}.gpr(k) + d^{(i)}.gpr(e) \cdot 4 \cdot size(t')$

↑ Softwaremultiplikation

Fall $R(v) = 0$
$R(v) = 1$: deref.

$v = e.n_i$

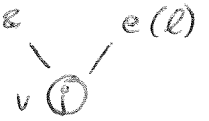


$type(e) = struct\{n_1 : t_1, \ldots n_s : t_s\}$

addi RD=j RS1=R imm = 4 · displ(i, type(e))

Fall $R(v) = 0$: v
$R(v) = 1$: deref

$v = \&e$



addi RS1 = L RD = j imm = 0

$R(e) = 0$
$R(v) = 1$

Übersetzung von Anweisungen

Zuweisung: $e = e'$

$R(e) = 0$
$R(e') = 1$

$j = code(e)$
$k = code(e')$

SW RD=k RS1=j imm=0

Verzweigung: if e then {a} else {b}

$k = code(e)$

- code(e)
- beqz RS1 = k, imm = length(code(a))+8
- code(a)
- j imm = -(length(code(a))+4+length(code(e))
- code(b)

Übersetzung von Ausdrücken

Modus 1: (R(e) = 1, "rvalue") liefert Wert eines Ausdrucks e

Modus 2: (R(e) = 0, "lvalue") liefert mit code(e) die Adresse des Ausdrucks e

Übersetzung von Anweisungen

Zuweisung: "e = e'" (R(e) = 0, R(e') = 1)

code(e) (mit Zielregister j)

code(e') (mit Zielregister k ohne Benutzung von Reg. j)

sw RD = k, RS1 = j, imm = 0

Verzweigung: "if e then {a} else {b}"

code(e) (mit Zielregister j)

beqz RS1 = k imm = length(code(a)) + 8

code(a)

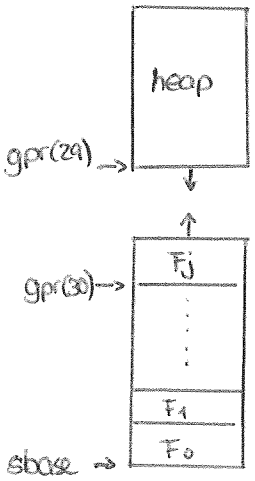
j imm = length(code(b)) + 4

code(b)

Funktionsaufruf: "e = f(e1, ... en)"

⚠ Einschränkung: type(e), type(ei) einfach ∀ 1 ≤ i ≤ n

Wiederholung: Layout von heap/stack



asize(c, Fj) = size(c.lms(j)) * 4 + 12

aba(c, Fj) = sbase + $\sum_{i<j}$ asize(c, Fi)

j > 0: d.m4(aba(c, Fj)) = ra return address

d.m4(aba(c, Fj)) + 4 = fb return bind address

d.m4(aba(c, Fj)) + 8 = pbase = (aba(c, Fj-1))

Stack und Heap disjunkt:

(*) d.gpr(30) + asize(c, top(c)) ≤ d.gpr(29)

Code: f aufrufende Funktion
f' aufzurufende Funktion

asize(f), asize(f') : Framegrößen

1. Abbruch, falls (k) verletzt

addi RS1 = 30, RD = 1, imm = asize(f) + asize(f')
subi RS1 = 1, RS2 = 29, RD = 1
sgri RS1 = 1, imm = 0, RD = 1

(gpr(1) = 0) $\Leftrightarrow$ gpr(30) + asize(f') + asize(f) − gpr(29) ≤ 0
$\Leftrightarrow$ aba(c, Fj+1) + asize(f) ≤ gpr(29))

beqz RS1 = 1, imm = 8
[Code für Abbruch: Instruktion, die das BS aufruft
⇒ siehe nächstes Kapitel]

2. Auswerten und Speichern der Parameter

∀ 1 ≤ i ≤ n: code(ei) (R(ei) = 1, Ziel-reg j)

sw RD = j, RS1 = 30, imm = asize(f') + 12 + 4 * (i − 1)

3. Initialisieren von phase in Fj+1

sw RD = 30, RS1 = 30, imm = asize(f') + 8

4. Ermittlung d. Adresse für den Rückgabewert

code(e) (R(e) = 0, Ziel-reg. j)

sw RD = j, RS1 = 30, imm = asize(f') + 4

5. Stackpointer erhöhen

addi RD = 30, RS1 = 30, imm = asize(f') ⚠ nicht vorher

6. Sprung zu code(f)

Sei a(f) Startadresse von code(f)

Erzeuge Code für 'gpr(j) = a(f)' (2 Instruktionen)

jalr RS1 = j

7. Initialisierung der return-Adresse

(am Anfang jeder Funktion)

sw RD = 31, RS1 = 30, imm = 0

⚠ a(f) für alle f erst nach Übersetzung bekannt

→ zwei Durchläufe: 1. Berechne a(f) mit offenen Sprungadressen
2. Trage Sprungadressen nach

⚠ Es erfolgt keine Initialisierung lokaler Variablen.

Für Simulation brauchen wir:

Softwarecondition: Keine Benutzung von lokalen Variablen VOR Assignment

Return „return e"

Einschränkung hier: Returntyp einfach

code(e) (R(e) = 1, Zielreg. k)

```
lw   RD = j, RS1 = 30, imm = 4     // lade rb für j+k
sw   RD = k, RS1 = j,  imm = 0
lw   RD = 31, RS1 = 30, imm = 0    // lade ra
lw   RD = 30, RS1 = 30, imm = 8    // lade alten Framepointer
jr   RD = 31
```

Allokierung "e = new(t)"

1. Abbruch, falls ⊛ verlegt wird

addi RS1 = 30, RD = 1, imm = asize(f) + asize(t)
// f akt. Fkt.

sub RS1 = 1, RS2 = 29, RD = 1

sgri RS1 = 1, RD = 1, imm = 0

beqz RS1 = 1, imm = 8

<Code für Abbruch>

code(e) (Zielregister k, R(e) = 0)

sub RS1 = 29, RD = 29, imm = asize(t)

sw RD = 29, RS1 = k, imm = 0

⚠ keine Garbage Collection (d.h. keine Freigabe von nicht erreichbarem Speicher)

Bsp: p = new (4);

p = new (4); // erster Wert nicht mehr erreichbar

Lücken / Hinweise:

Nicht abgefangene Fehler ...

→ im CO-Programm

Nichtterminierung, Nullpointer deref., Array out of index bounds

→ bei Compilierung

Code size restrictions wg. 16 Bit Imm

→ Ausführung des comp. Programms

out of memory wg. stack / heap overflow

→ Init / Exitcode

Kapitel: Betriebssystemkerne

Betriebssystemkerne erlauben kontrollierten, pseudoparallelen Ablauf von mehreren Nutzerprogrammen.

Übersicht: → Betriebssystemunterstützung in Prozessoren

- Interrupts / Special purpose register
- System mode / user mode
- Memory Managementunit (MMU) u. Adreßübersetzung
- In / Output Devices

→ Communicating Virtual Machines (CVM)

- abstraktes Berechnungsmodell für
  - einen abstrakten Kern k geschrieben in CO
  - mehrere Nutzerprogramme geschrieben in DLX-Assembler

→ Implementierung

- abstrakter Kern k wird durch konkreten Kern K implementiert
- hierzu benötigt man:
  Erweiterung von CO: COA unterstützt Inline Assembler Anweisungen f. hardwarenahe Programmierung

Betriebssystemunterstützung für Prozessoren

Interrupt: Ein Ereignis, das den „normalen" Programmablauf unterbricht und stattdessen eine Interrupt service routine (ISR) aufruft.

Grober Ablauf der ISR

1. Interrupt wird ausgelöst
2. CPU speichert: - Teile des akt. Programmzustands (insb. PC)
   - Quelle des Interrupts
3. Behandlung des Interrupts
4. Heißens: Rückkehr ins unterbrochene Programm

Klassifikation

1. Externe / interne Interrupts
   - ↳ ausgelöst vom Gerät, bspw. Tastendruck
   - ↳ ausgelöst v. Programm
2. Maskierbare / nicht maskierbare Interrupts
   - ↳ können durch SW abgeschaltet werden
3. Resume Type

   Sei I unterbrochene Instruktion, I' Instruktion, die ohne Interrupt folgen würde

   repeat: nach ISR mit I weitermachen

   continue: nach ISR mit I' weitermachen

   abord: weder noch

4. Priorität: regelt welcher Interrupt bei gleichzeitigem Auftreten zuerst behandelt wird

   iev(j): internal event

   Interner Interrupt vom Typ j, $j \in \{1, .. 6\}$

   eev(j): external event

   Externer Interrupt vom Typ j, $j \in \{0, 7-31\}$

Neues Argument der next-state Funktion δ: $\delta(c, eev) = c'$

Übersicht:

j	typ	resume	maskierbar	extern	Beschreibung
0	res	abort	nein	ja	restart
1	ill	abort		nein	illegal
2	mis	abort			misalignment
3	pff	repeat			page fault on fetch
4	pfls	repeat			page fault on load/store
5	trap	continue			trap
6	ovf	continue	ja		overflow
≥7	I/O	continue	ja	ja	Ein/Ausgabe

5.7.2006

event signale

eev[0] input
eev[1] } innerhalb CPU
⋮
eev[6] }
eev[7] } input
⋮
eev[31] }

Vektor externer event-Signale:

$eev = (eev[0], eev[7], \ldots eev[31])$

Cause Signal

$$CA(c,eev)[j] = \begin{cases} eev[j] : j \in 0,7..31 & \leftarrow \text{Konfigurationsunabhängig} \\ rev[j] : j \in 1..6 & \leftarrow \text{Konfigurationsabhängig} \end{cases}$$

Masked Cause

$$MCA(c,eev)[j] = \begin{cases} CA(c,eev)[j] & j > 5 \\ CA(c,eev)[j] \wedge c.SR[j] & j \leq 5 \end{cases}$$

↑ status register

weitere benutzersichtbare Register: Special purpose register file $c.SPR:[0,1]^5 \to [0,1]^{32}$

<i>	SPR(i)	
0	SR	status register
1	ESR	exception status reg.
2	ECA	exception cause
3	EPC	exception PC
4	EDATA	exception data

$JISR(c,eev) = \bigvee_j MCA(c,eev)[j]$ jump to interrupt service routine

$SISR = 0^{32}$ start address interrupt service routine (in RCH, da nicht flüchtiger Sp.)

Fall: $JISR(c,eev) = 0$ $\Rightarrow \delta_{neu}(c,eev) = \delta_{alt}(c)$

$il(c,eev) = \min\{j \mid MCA(c,eev)[j] = 1\}$ interrupt level

kleinste Interruptnummer hat höchste Priorität

$JISR(c,eev) = 1$ $\Rightarrow c' = \delta_{neu}(c,eev)$ Effekt des Interrupts

- $c'.SR = 0^{32}$ maskiere alle Interrupts, da Interrupt = „Art von" Funktionsaufruf Handler ist

  Implementierung hiervon soll NICHT unterbrochen werden

- $c'.EPC = \begin{cases} c.PC & : il(c,eev) \in \{3,4\} \text{ repeat} \\ \delta_{alt}(c).pc & : \text{sonst} \quad \text{continue} \end{cases}$

  Rücksprungadresse nach Ausführung des Handlers

- $c'.EDATA = \begin{cases} sirimm(c) & : il(c,eev) = 5 \\ ea(c) & : il(c,eev) = 4 \end{cases}$ pfls — Konst. aus hap Instr. ↓ neue Instruktion

- $c'.PC = 0^{32} = SISR$

- $c'.ESR = \begin{cases} c.gpr(RS1(c)) & : \overbrace{movi2s(c) \wedge (SA(c) = 0^5)}^{\text{setzt GR}} \wedge \overbrace{(il(c,eev) \geq 5)}^{\text{continue}} \\ c.SR & : \text{sonst} \end{cases}$

- $c'.ECA = MCA(c,eev)$

4 neue Instruktionen:

R-type

opcode	RS1	RS2	RD	SA	Fu

movs2i
movi2s

010000 $c'.gpr(RS1(c)) = c.spr(SA(c))$
010001 $c'.spr(SA(c)) = c.gpr(RS1(c))$

Datentransfer zwischen general und special purpose register file

I-type

opcode	imm

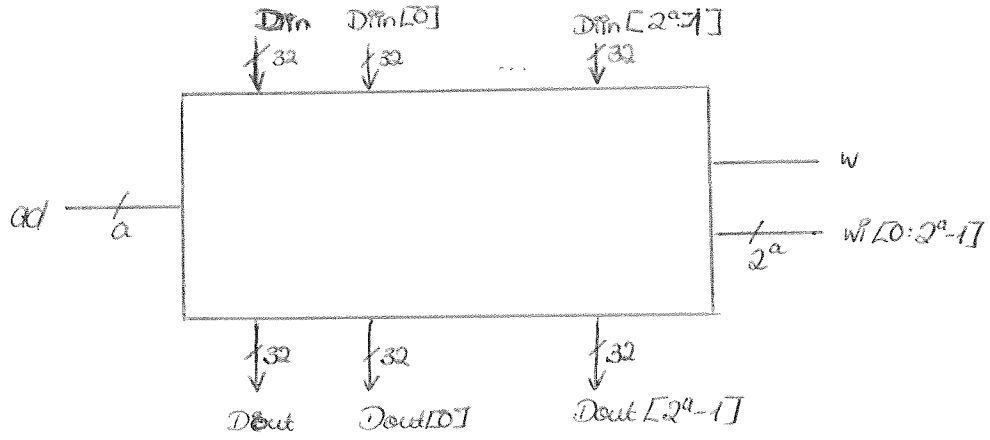
trap 111110 $rev(c)[5] = 1$

rfe 111111 $c'.PC = c.EPC$, $c'.SR = c.ESR$ } restauriert PC und status reg.

↑ return from exception

Hardware:

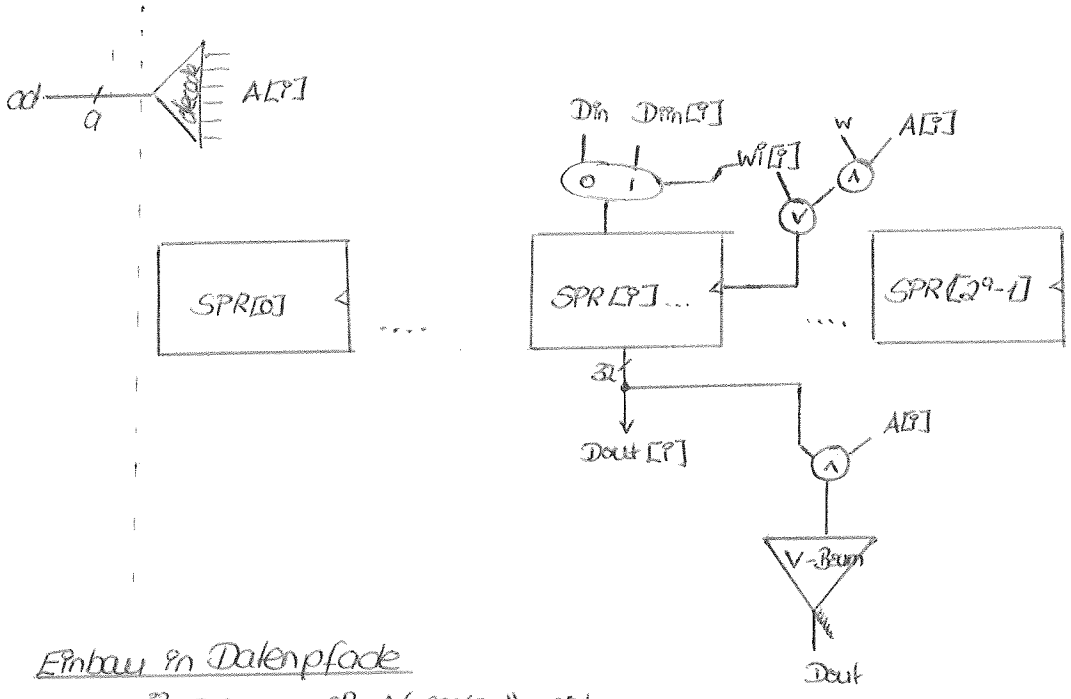
$h.spr : \{0,1\}^5 \to$



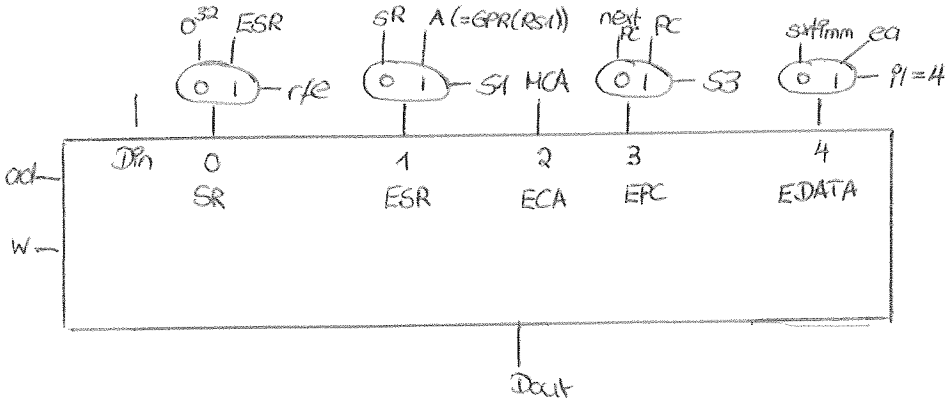
$Dout[i](h) = h.spr[i]$

$Dout(h) = h.spr(adr(h))$ RAM

$$h'.spr(i) = \begin{cases} Din[i](h) : & wi[i](h) \\ Din(h) : & \neg wi[i](h) \wedge w(h) \wedge adr(h) = i \\ h.spr(i) : & sonst \ (speichern) \end{cases}$$



Einbau in Datenpfade



$S1 = (pl \geq 5) \wedge movi2s \wedge SA = 0^5$

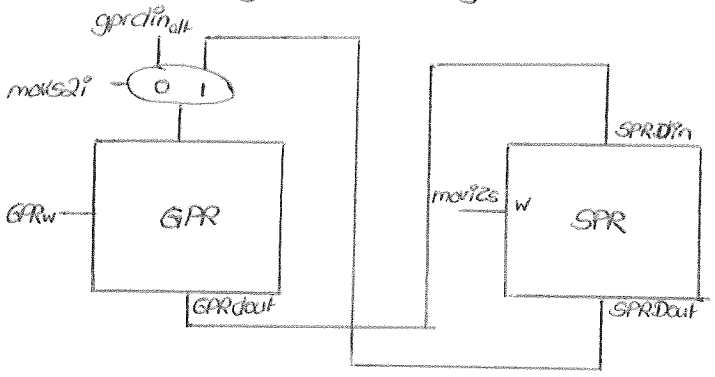
$S3 = 1 \Leftrightarrow$

$pl \in [3,4] = MCA[3] \vee MCA[4] \wedge \overline{\bigvee_{j \geq 2} MCA[j]}$

Aktivierung der write-individual Signale wifn

wifn[0] = rfe(c) ∨ JISR(c,eev)	SR
wifn[1] = JISR(c,eev)	ESR
wifn[2] = JISR(c,eev)	ECA
wifn[3] = JISR(c,eev)	EPC
wifn[4] = JISR(c,eev)	EDATA

Implementierung: Erweiterung der Datenpfade



write Signale

SPR.w = ex ∧ movi2s ∧ (pl ∈ {2,3,4})

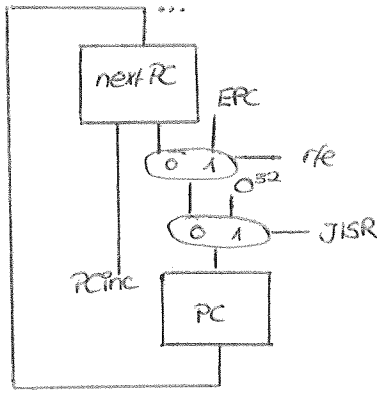
Beachte: kein SPR-update bei Interrupt vom Typ repeat/abort!

GPR.w_alt = ex ∧ (lw_h ∨ comp_h ∨ compi_h ∨ jal_h ∨ jalr_h)

GPR.w = ex ∧ ((lw_h ∨ comp_h ∨ compi_h ∨ jal_h ∨ jalr_h) &∨ movs2pi_h) ∧ (pl(c) ∉ {2,3,4})

mw = mw_alt ∧ (pl ∉ {2,3,4})

Program Counter bei JISR/rfe



Implementierung der fev[j] (parallel) j ∈ {1..6}

fev[1] = ¬ (lw ∨ sw ∨ ....) ∧ ex
(lw ∨ sw ∨ .... : Instruction decoding Prädikate)

fev[2] = mal = pmal ∨ dmal
(pmal: instruction; dmal: data misalignment)

pmal = $\overline{ex}$ ∧ (PC[0] ∨ PC[1]) , PC[1:0] ≠ 0²
dmal = ex ∧ (lw ∨ sw) ∧ (ea[0] ∨ ea[1])

iev [3] = pff } kommt nach
iev [4] = pfls }
↑ Register

iev [5] = trap ∧ ex trap: Decoding-Prädikat für trap-Instruktionen

iev [6] = ovf = ALUovf ∧ (addo ∨ subo ∨ addfo ∨ subfo)

Add/sub Instruktionen mit overflow

Virtual Memory Support

Virtual Memory: Simulation von großem RAM-Speicher durch kleinen RAM-Speicher und Festplatte (HardDisk)

Ziel: Simulation von vielen (großen) RAM Speichern zur Ausführung vieler Nutzerprogramme

Demand Paging: - Page/Seite: Speicherbereich von 4K Größe

- „Paging": Transfer von Seiten zwischen swap Memory (→ HardDisk) und (Physical) Memory (→ RAM)
- „on demand": Seite wird (spätestens) dann geladen, wenn sie vom Nutzerprogramm benötigt wird.

Physikalische/virtuelle Maschinen:

virtuelle DLX Maschine = bisherige DLX Maschine

→ modelliert (später) Ausführung eines Nutzerprogramms

physikalische Maschine = bisherige DLX Maschine + MMU/Adreßübersetzung

→ erlaubt viele virtuelle Maschinen zu simulieren

→ Adressen der virtuellen Maschine heißen virtuelle Adressen, die der physikalischen Maschine heißen physikalische Adressen

Speicher der virtuellen Maschine wird in Seiten der Größe 4K unterteilt

$vpage(v.px) = m_{4096}(v.px\ 4096)$

virtual page index $\in \{0,1\}^{20}$ byte index $\in \{0,1\}^{12}$

Spezifikation der physikalischen Maschine

Bemerkung: physikalische Maschine ist nur für den Betriebssystemkern sichtbar

über neue SPRs:

?	Name	Bedeutung	
5	PTO	page table origin	} Register
6	PTL	page table length	
7	MODE	mode	
8	EMODE	exception mode	

Prozessor unterstützt zwei Modi:

Falls $MODE = 0^{32}$: Systemmodus — Ausführung des Betriebssystems

$MODE = 0^{31}1$: Usermodus — für die Ausführung von Nutzerprogrammen

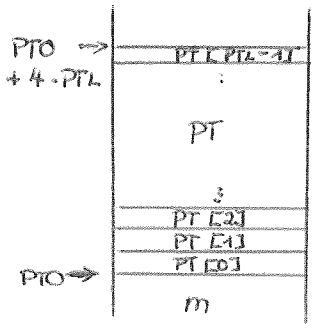
Usermode:
- Adreßübersetzung
- kein movr2s, movs2r

$\rightsquigarrow ev[1] = ev_{alt}[1] \vee (ex \wedge (movr2s \vee movs2r) \wedge MODE = 1)$

Adreßübersetzung: für va = PC oder va = ea

- Übersetzung von va erfolgreich: va → pa
- Übersetzung nicht erfolgreich → page fault

va = pc → pff
va = ea → pfls

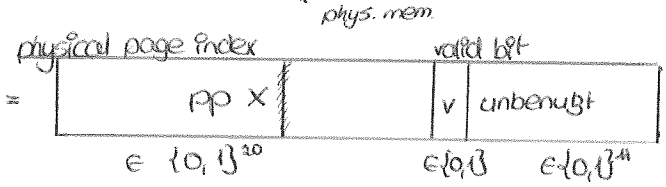


page table origin PTO
page table length

$va = (va.px, va.bx)$

page table entry address: $ptea(c, va) = c.PTO +_{32} (va.px, 00)$

page table entry: $pte(c, va) = c.pm_4(ptea(c, va))$ (phys. mem.)

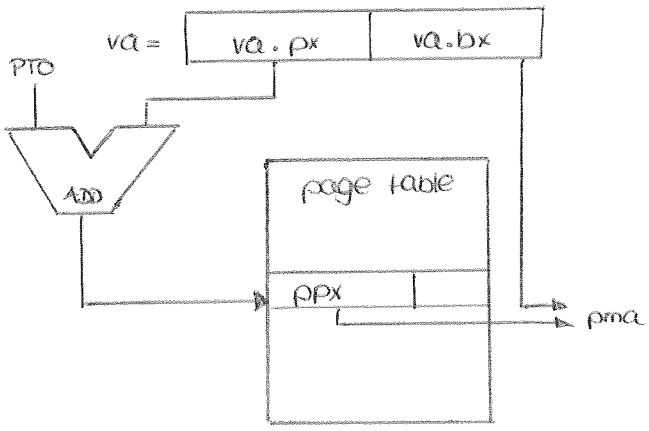


$ppx(c, va) = pte(c, va)[31:12]$

$v(c, va) = pte(c, va)[11]$

$pma(c, va) = (ppx(c, va), va.bx)$

physical memory address: wird für den Zugriff auf das Physical Memory verwendet, wenn kein page fault auftritt



Page Fault

- falls va.px außerhalb der page table ⇒ Page Table Length Exception

  $ptlef(c) = \langle c.PC.px \rangle \geq \langle c.ptl \rangle$ (on fetch)

  $ptlels(c) = \langle ea(c).px \rangle \geq \langle c.ptl \rangle$ (on load/store)

- falls valid-Bit = 0 ⇒ Page Table Entry Exception

  Bedeutung: Seite (momentan) nicht im Physical Memory, sondern im swap

12.07.2006

Wiederholung:

Virtual Memory Support / Physikalische Maschine

Spezifikation:

- Neue SPRs: Page table origin (5), page table length (6), Mode (7), EMode (8)
- Speicher: $c.pm : \{0,1\}^{32} \to \{0,1\}^8$ (physical memory)

  $c.sm : \{0,1\}^{37} \to \{0,1\}^8$ (swap memory)

  30+7 ⇝ 128GB (128 GB)

  (c.sm ist Abstraktion und wird durch Hard Disk od. anderes Gerät implementiert)

Definition: $\delta(c, eev) = c'$

Fall $c.MODE = 0$ (Systemmode)

wie vorher

Fall $c.MODE = 1$ (Usermode)

- Illegal Exception bei mvrs2i, mvi2s
- Adreßübersetzung

Mode Changes: Nur über Interrupts oder durch Betriebssystem

- $JISR(c, eev) = 1$ zusätzlich

  $c'.MODE = 0$
  $c'.EMODE = c.MODE$

- Bei rfe zusätzlich $c'.MODE = c.EMODE$

  (Direktes Beschreiben von MODE möglich, aber meist nicht gewollt)

Adreßübersetzung (für $MODE = 1$)

→ virtuelle Adresse $va = va.px, va.bx$
$\in \{0,1\}^{32}$ $\in \{0,1\}^{20}$ $\in \{0,1\}^{12}$

→ $ptea(c, va) = c.PTO + 32 \cdot (0^{10}, va.px, 00)$

$pte(c, va) = c.pm_4(ptea(c, va))$

$ppx(c, va) = pte(c, va)[31:12]$

$v(c, va) = pte(c, va)[11]$

$pma(c, va) = (ppx(c, va), va.bx)$

Page Fault (bei Zugriff außerhalb der Page Table oder invalid Page Table entry)

$ptlef(c) = \langle c.PC.px \rangle \geq c.PTL$
$ptels(c) = \langle ea(c).px \rangle \geq c.PTL$

$iev(c)[3] = pff(c) = c.MODE \wedge (ptlef(c) \vee \neg v(c.PC))$

(Falls $iev(c)[3] = 0$: $I(c) = \begin{cases} c.pm_4(c.PC) & : \text{falls } c.MODE = 0 \\ c.pm_4(pma(c, c.PC)) & : \text{falls } c.MODE = 1 \end{cases}$)

$iev(c)[4] = pfls(c) = \neg pff(c) \wedge c.MODE \wedge (I(c) = lw \ldots \vee I(c) = sw \ldots) \wedge (ptels(c) \vee \neg v(c, ea(c)))$

(Falls $iev(c)[3] = 0$ und $iev(c)[4] = 0$, dann benutze $pma(c)$ für etwaigen Memory Zugriff)

Implementierung

SPR, JISR, rfe - Änderungen: nicht gezeigt

Translation - Phases:

Vorher nur ex Register für Fetch Phases (ex=0) und Execute-Phases (ex=1)

Zusätzlich:

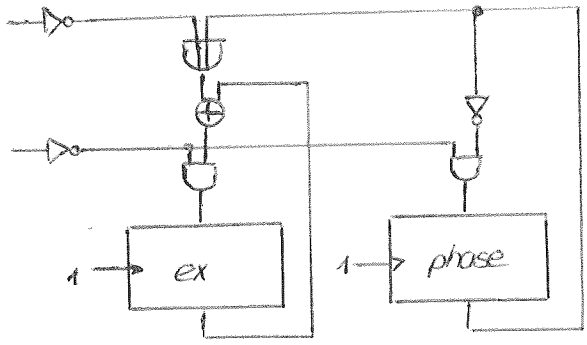
Register phase zur Unterscheidung von Address-Translation (phase=0)

und Memory Access (phase=1)

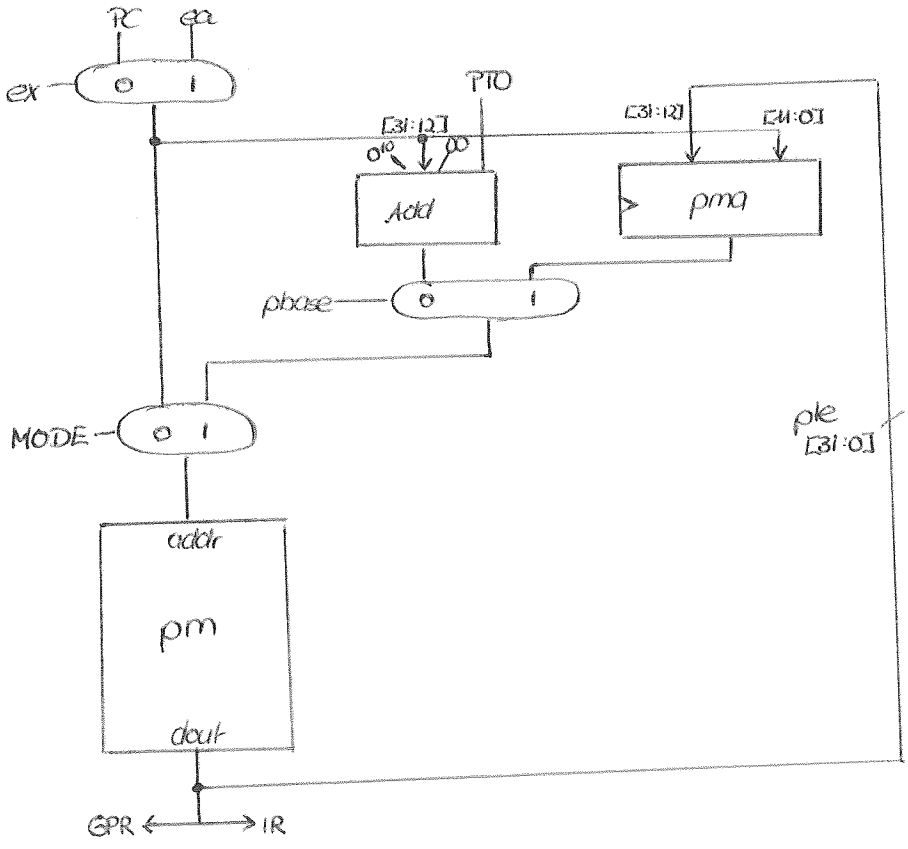
User Mode:

ex und phase arbeiten wie ein 2-Bit Zähler

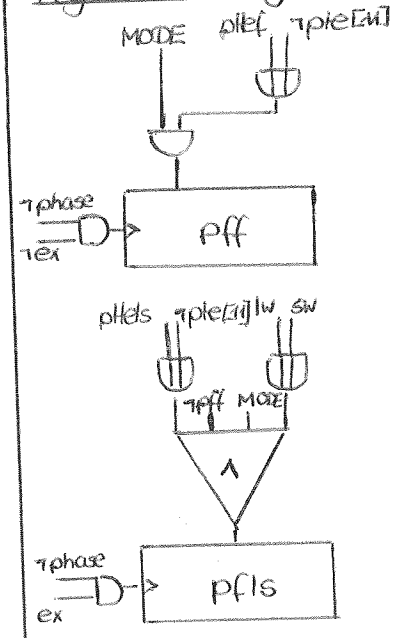
	ex	phase	Beschreibung
t	0	0	Translate PC
t+1	0	1	Fetch Instruction word
t+2	1	0	Translate ea
t+3	1	1	Access ea



Memory Access



Page Fault Flags



Communicating Virtual Machines (CVM)

Neues Maschinenmodell

Bestandteile: - der abstrakte Kern k
(CO-Programm, läuft später im Systemmodus)
- virtuelle Maschinen vm(p) (Benutzerprogramme)
- I/O Devices D(j)

CVM Konfiguration: $cvm = (cvm.c, cvm.vm(1) \ldots cvm.vm(p), cvm.pH(1) \ldots cvm.pH(p), cvm.cp)$

- cvm.c : Konfiguration von k
- Benutzerprogramm $p \in \{1..p\}$ (p = maximale Anzahl v. Benutzerprogrammen)
- Konfiguration cvm.vm(p) der virtuellen Maschine
- Speichergröße cvm.pH(p) in pages, d.h. $cvm.vm(p).vm : va(p) \to \{0,1\}^s$

$va(p) = \{a \in \{0,1\}^{32} \mid 0 \leq a < cvm.pH(p) \cdot 4096\}$

- current process: $cvm.cp = 0$ ⇒ abstrakter Kern "läuft"
$cvm.cp \neq 0$ ⇒ Benutzerprogramm cvm.cp "läuft"

Next state Funktion von CVM

$\delta(cvm) = cvm'$

- $u = cvm.cp > 0$, $JISR(cvm.vm(u)) = 0$

$cvm'.vm(u) = \delta_{Dux}(cvm.vm(u))$ Benutzer führt Schritt aus
$cvm'.vm(u') = cvm.vm(u')$ für alle $u' \neq u$
$cvm'.c = cvm.c$
$cvm'.cp = cvm.cp$

- $cvm.cp = 0$: $cvm.c.pr = an;$ ↯ (Kern führt Schritt aus - keinen „besonderen")

Fall: an keine „besondere Funktion"

$cvm'.c = \delta_{co}(cvm.c)$
$\forall u \in \{1..p\}: cvm'.vm(u) = cvm.vm(u)$
$cvm'.cp = cvm.cp$

- $cvm.cp = 0$ $cvm.c.pr = an;$ ↯

$an = start\,cp$ //startet Benutzerprogramm, das durch cp von k bestimmt wird

$cvm'.cp = va(cvm.c, CP)$ („Scheduler": legt für upcoming von CP aus fest)

CVM (communicating virtual machines)

cvm Konfiguration

Komponenten:

- cvm.c : CO Konfiguration, abstrakter Kern
- cvm.vm(u): $u \in [1..p]$ User machines, DLX-Konfiguration
- cvm.pH(u): page table length für Benutzer u (Speicher für Ben.u)
- cvm.cp $\in [0..p]$: current process

$$cvm.cp = \begin{cases} 0 & \text{Kern läuft} \\ u > 0 & \text{Benutzer u läuft} \end{cases}$$

- cvm.d(?) : Zustand von Device ?

$\delta(cvm) = cvm'$

1) cvm.cp = u > 0 (JISR (cvm.vm(u)))

→ $\boxed{cvm'.vm(u) = \delta_{DLX}(cvm.vm(u))}$

Ein Schritt Benutzer u, VH
Implementierung evtl. page faults

cvm'.vm(?) = cvm.vm(?) ?≠u
cvm'.cp = u
cvm'.c = cvm.c
cvm'.d(?) ⚠

2) cvm.cp = 0 cvm.c.pr = an; t an keine spezielle Funktion

→ $\boxed{cvm'.c = \delta_{CO}(cvm.c)}$

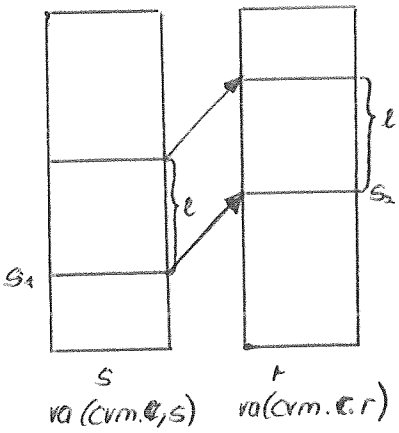
$cvm'.vm(u) = cvm.vm(u) \quad \forall u$

3) cvm.cp = 0 cvm.c.pr = an; t an ist spezielle Funktion

- an = startcp Prozess 'CP' starten

$\boxed{cvm'.cp = va(cvm.c, CP)}$

- an = copy (s, r, s1, s2, l)



$cvm'.vm(\underbrace{va(cvm.c,r)}_{\text{receiver}}).m_{va(cvm.c,l)}\ (va(cvm.c,s2))$

$= cvm.vm(va(cvm.c,s)).m_{va(cvm.c,l)}\ (va(cvm.c,s1))$

- an = alloc (u, x)

neue Seiten für Benutzer va (cvm.c, u)

cvm'.ptl (va (cvm.c, u)) = cvm.ptl (va (cvm.c, u)) + va (cvm.c, x)

- an = free (u, x)

entfernt die letzten x pages von Benutzer u

cvm'.ptl (va (cvm.c, u)) = cvm.ptl (va (cvm.c, u)) − va (cvm.c, x)

Aufruf des Kerns vom Benutzer aus

(Binäres Interface des Kerns) durch trap Instruktionen

cvm.cp = u > 0 trap (cvm.vm(u))

Sei f = [sxffmm (cvm.vm (u))]

Wir rufen Funktion ⊛ kcal (f) ∈ fname ← Funktionsnamen des Kerns

↑ kernel call definition function

festes Parameter des Modells

Sei n = 10, n: # Parameter von f

⊛⊛ Wir übergeben als Parameter Register [1:10] des Benutzerprocesses

cvm'.cp = 0

Funktionsaufruf Semantik, von Benutzer u aus

cvm'.c.rd = cvm.c.rd + 1 neuer frame cvm'.c.pr = f + (f).body ; cvm.c.pr top(cvm'.c).el(l) = cvm.vm(u).gpr(l) $\underbrace{\text{wert Param. l}}$ $1 \le l \le u$

trap ≙ formal (remote) function call

Kern Implementierung

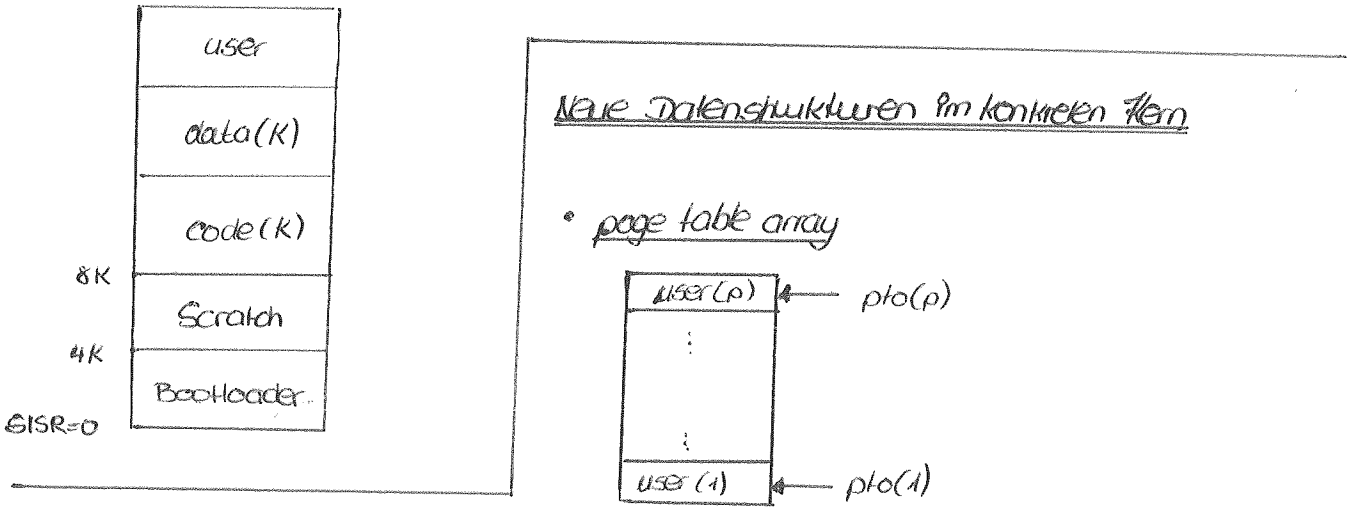
durch konkreten Kern K

- hat Inline Assembler Code
- wird um abstrakten Kern "herumgeschickt" (theory of linking)

Inline Assembler

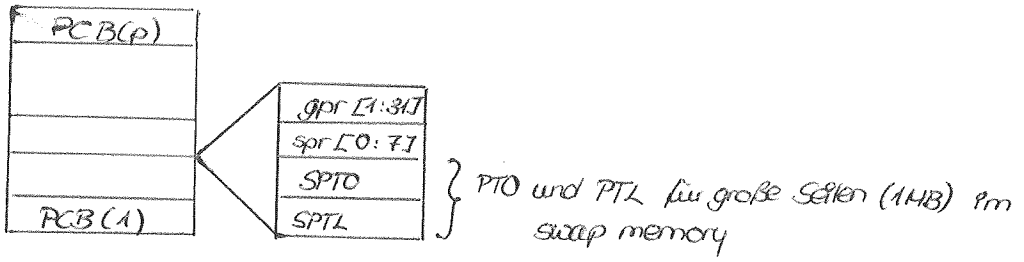
- Syntax : asm(u) u Folge von DLX Instruktionen
- Compilieren: code(asm(u)) = u
- Semantik: ??

Memory Map der physikalischen Maschine



- process control blocks (PCB)

PCB(u) PCB von Prozess u



- swap memory page table SPT
- 4 gelinkte Listen:

→ used list	: benutzte Seiten im phys. memory	} von alloc, free, pf-handler benutzt
→ free list	: freie Seiten im phys. memory	
→ swap freelist (sfree)	: freie Seiten im swap memory	
→ swapusedlist (sused list)	: benutzte Seiten im swap memory	

Eine große Seite reservieren wir im swap-memory durch alloc (, x)

$x \cdot 4K = 1M \qquad x = \frac{1M}{4K} = 256$

y große Seiten : $X = Y \cdot 256$

Eur Invariante des Korrektheitsbeweises

Konfigurationen $d^1, d^2 \ldots$ der physikalischen Maschine,
$k^1, k^2 \ldots$ des konkreten Kerns (formal: CO-Konfigs),
$cvm^1, cvm^2 \ldots$ der CVM Maschine,
$aba^1, aba^2 \ldots$ allocated base address des konkreten Kerns,

für globale Variablen (ins PCB, PT, SPT) aba konkret = aba^0 (konstant)

Definiere • für Ausdrücke e, typ(e) einfach nur mit globalen Variablen

• für physikalische DLX Maschine d

$$\boxed{va(d,e) = d.m_4(aba^0(e))}$$

$pto(d,u) = va(d, PCB(u).pto)$ ↖ spr()

$ptea(d,u,va) = pto(d,u) + 4 \cdot va.px$ $\qquad va = va.px, va.bx$

$pte(d,u,va) = d.m_4(ptea(d,u,va))$

$ppx(d,u,va) = pte(d,u,va)\ [31:2]$

$v(d,u,va) = pte(d,u,va)\ [1:1]$

$$\boxed{pma(d,u,va) = ppx(d,u,va) \circ va.bx}$$

Analog:

sptea, spte, sma ...

Ebene	CO-Konf.	Sprache
abstrakt	c	CO
konkret	k	CO_A

cvm : cvm.c, cvm.vm(u), cvm.cp ...

Implementierung - Korrektheit : 3 Rechnungen

cvm^i $i = 0, 1, 2$
k^i
d^i

physikalische DLX-Maschine

sim(cvm, k, d)

- d kodiert cvm.vm(u) $\forall u$ und k ← consis() klaus Compilerkorrektheit
- k kodiert cvm.c
  k consis

e : Ausdruck mit ausschließlich globalen Variaben einfachen Typs

var(d, e) = d.m_4(aba°(e))

pto(d, u) = va(d, PCB[u].pto ...)
↖ spr()

ptea(d, u, va) = pto(d, u) + 4 · va.px Pointer-Arithmetik!

ppa(d, u, va)
ppx

sma(d, u, va)
v

Definition : B-Relation

B(cvm, u, d) ≡ d kodiert cvm.vm(u)

- Register :

$$cvm.vm(u).R = \begin{cases} d.R & : cvm.cp = u \\ va(d, PCB[u].R & : cvm.cp \neq u \end{cases}$$

Spezifikation für den Prozeßwechsel

- Speicher:

$$cvm.vm(u).m(va) = \begin{cases} d.m\ (pma(d,u,va)) : & v(d,u,va) = 1 \\ d.sm\ (sma(d,u,va)) : & sonst \end{cases}$$

Spezifikation vom pf handler + ⊛

⊛ globale Variable MRL most recently loaded page

pf: Fall: free-list ist leer ⇒ keine freien Seiten im phys. Memory d.m

eine Seite (u', vpx) : victim page index in swap Memory zurückschreiben

$(u', vpx) \neq MRL$

Grund: eine Instruktion kann 2 page faults auslösen:
- pff, dann pfls
- pfls, dann pff

Definition: sim

sim (cvm, k, d, aba ...)

- $\forall u\ B(cvm, u, d)$
- consis (k, aba, d)
- kconsis (k, aba, cvm.c) // konkreter u. abstrakter Kern sind konsistent

kalloc: {V | V Variable von cvm.c} → (V ... k)

ausdehnen auf Subvariablen:

e-konsis: V elementare Subvariable von cvm.c

va (cvm.c, V) = va (k, kalloc (V))

p-konsis: Pointerkonsistenz

va (cvm.c, V) = LL Subvariable, d. h. V Pointer

va (k, kalloc (u)) = kalloc (u) für erreichbare V

Korrektheit der cvm-Implementierung

$\exists$ Folgen von Schrittzahlen $s(1), s(2), s(3)\ldots$
$t(1), t(2), t(3)\ldots$

$\exists$ Folgen von Konfigurationen k^i, d^i, $i = 0,1,\ldots$

$\exists$ Folgen von aba^i, $kalloc^i$, so daß für alle n gilt:

$$sim(cvm^n, k^{s(n)}, d^{t(n)}, aba^n, kalloc^n)$$

Induktionsschritt: $sim(cvm^n, k^s, d^t, aba^n, kalloc^n)$

$\exists s\ \exists t$: $sim(cvm^{n+1}, k^{s+\sigma}, d^{t+\tau}, aba^{n+1}, kalloc^{n+1})$, so daß

Implementierung

Fälle: aus cvm spec

① • $cvm^n.cp = u > 0$ $trap(cvm^n.vm(u), i)$

• $cvm^n.cp = u > 0$ $JISR(cvm^n.vm(u))$

• keine page faults in $d^{t(n)}$ ⇐ spec von DLX mit MMU

② • $JISR(d^{t(n)})$ page fault

ad ① ISR: teste $i = 0$ (reset)

ja: bootloader
nein: rette Register d. R von $vm(u)$ in PCB[u]
↑ muß in assemblercode gemacht werden

$? u = va(d^{t(n)}, CP) = va(k^{s(n)}, CP)$ mit $consis(k^{s(n)}, aba, d^{t(n)})$

⚠ nicht in code(k)

teste ECA, geht nicht direkt, da SPR

• movs2i in GPR(1)

evtl. kein reset: rette vorher GPR(1) in scratch ∉ ROM

Bestimme il (Interrupt level) durch C-Code

° untersuche PCB[CP].ECA

$va(k^{s(n)+5'}, \ldots) = cvm^{n+1} \cdot vm(u).eca$ (Korrektheit save)

Fall: $il = min\{j \mid eca(j) \neq 0\} = 5$

Bestimme f = PCB[CP].EDATA

geschachtelte if-then-else

Aufruf von f = kcd(f) n = # Parameter von f

C-function call f(PCB[CP].gpr[1:n])

Rest: Korrektheit von Compiler

Fall: function call

Semantik von asm(u)

Euler Rechnungen:

$k^{s(n)}$ $k.pr = asm(u);r$

$d^{t(n)} \to d^{t(n)+1} \ldots d^{t(n)+t'}$ durch Ausführung von u

$\downarrow \quad \downarrow$

$k^{s(n)} \; \# \# \; \tilde{k}^1 \ldots \tilde{k}^{t'-1} \; \tilde{k}^{t'} = k^{s(n)+1}$

$\tilde{k}^{i+1} = \tilde{k}^i$, falls $sw(d^{t(n)+i})$ und $\exists$ C-Variable X mit $ea(d^{t(n)+i}) = obj(X)$

Falls X nicht global $\to$ Fehler

Wert von $\tilde{k}$: $va(\tilde{k}^{i+1}, X) = d^{t(n)+i}.gpr(RD(d^{t(n)+i}))$